Using Data Clustering to Improve Cleaning Performance* for Flash Memory

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, Taiwan, ROC

SUMMARY

Flash memory offers attractive features for storage of data, such as non-volatility, shock resistance, fast access speed, and low power consumption. However, it requires erasing before it can be overwritten. The erase operations are slow and consume comparatively a great deal of power. Furthermore, flash memory can only be erased a limited number of times. To manage flash memory effectively, we propose in this paper a new approach to reduce the number of erase operations performed. Via the accessing frequency of data, data is dynamically classified and clustered together to reduce the amount of data copied and the number of erase-operations. In addition, an adaptive cleaning manager is designed to dynamically adjust cleaning policies in response to the variations of data access behavior. Performance evaluations through real implementation on a flash memory server and trace-driven simulation show that the number of erase operations was reduced by 16~22.6%, the number of blocks copied was reduced by 20.1~28.4%, and throughput was improved by 19.5~25.2%. Even wearing is also ensured.

KEY WORDS: flash memory; cleaning policy; mobile computer; consumer electronics;

INTRODUCTION

Flash memory shows promise in many applications as storage media due to its fast access speeds, low power consumption, non-volatility, and rugged operations.[1-6] Besides, flash memory is small and lightweight, so it is widely used in consumer electronics, such as digital cameras, audio recorders, cellular phones, as well as in embedded systems and mobile computers, such as notebooks, handheld computers, personal digital assistants (PDAs),[7,8] etc.

Unlike RAM and hard disks, flash memory has special hardware characteristics which must be dealt with.[3,9,10] Flash memory is partitioned into segments[†] that are fixed by the hardware manufacturers. It cannot be written over existing data unless the flash segment is erased in advance. Erase operations are slow (typically 0.6~0.8 seconds), waste relatively lots of power

---

* This work is a part of *RAMOS* project in IIS, and is submitted to *Software: Practice and Experiences* for publications.

* We use "segment" to represent hardware-defined erase block and "block" to represent software-defined block.

and can only be performed on whole segments. Furthermore, the number of times a segment can be erased is limited (typically 100,000~1,000,000 times). Because of these hardware limitations, flash memory-based storage systems should avoid erasing as much as possible in order to prolong flash memory lifetime, increase system performance, and conserve power. The write and erase operations should be operated evenly over the whole flash memory to avoid wearing out some segments, which adversely affects the usefulness of flash memory.

Because segment sizes (typically 64 Kbytes) are much larger than the block sizes that storage systems typically use (i.e. 1Kbytes~8Kbytes), to avoid having to erase, systems that have large segment sizes do not update data in place.[5,6,11-13] Updates are written to any empty flash memory space and obsolete data are invalidated as garbage. A software *cleaner* later reclaims garbage by migrating valid data from the segment to be cleaned to another segment and then erasing the original segment. With this *non-in-place-update* mechanism, the cleaner has significant effect as the utilization (the percentage of flash memory space occupied by valid data) gets higher. More segments need to be reclaimed in order to have one free segment, more data are migrated and more erasures must be done. Performance is thus severely degraded, lifetime is greatly decreased, and energy consumption is greatly increased, as is clearly demonstrated in the literature.[4,5]

Since the cleaner has a substantial impact on energy consumption, performance, and endurance, cleaning policies that control which segments to clean, when to clean them, and how to clean them can severely affect cleaning performance.[5,6,12] Two major concerns regarding to the clean policies are the *segment selection algorithm* that determines the segments to be cleaned and the *data redistribution method* (or *data reorganization method*) that determines how to migrate valid data in the selected segments. The data redistribution method, however, has the most important impact on cleaning performance, as shown in a previous study.[12]

The way valid data are migrated during segment cleaning can severely affect future cleaning costs. The simplest way is to copy valid data to another segment in the same order as they appear in the original segment. If data are migrated in the way that *hot data* (most frequently updated data) are clustered in the same segments, then flash segments will be either full of all hot data or all non-hot data. Because hot data have high possibility to be updated soon, the original copy soon becomes garbage. Therefore, segments containing most of the hot data would soon contain the largest amount of garbage. Cleaning these segments then can reclaim the largest amount of garbage and the least valid data must be migrated during cleaning. Cleaning costs thus can be significantly reduced. Since separately clustering hot data and cold data can reduce cleaning overhead, the major problem of cleaning becomes how to effectively cluster hot data. Solving this problem is one significant aim of this paper.

In this paper, we propose a new approach that dynamically reorganizes data in a fine-grained and light-overhead way. Data are classified according to their write access frequencies and are clustered during data updating and during segment cleaning, without the complex computations for determining data as hot or cold. Another aim of this paper is to apply an adaptive approach to dynamically adjust policies in response to the variations of data write access behaviors This is motivated by the fact that no single cleaning policy performs well across a wide variety of workloads and access patterns.

In order to demonstrate the advantage of the proposed methods, a flash memory server and a trace-driven simulator are implemented with various cleaning policies and various data

redistribution methods. Performance evaluations with various cleaning policies show that the proposed data clustering method significantly reduced 11.5~22.6% of erase operations and 14.4~28.4% of blocks copied, and improved average throughput by 12.2~27.1%. The adaptive cleaning policy performed better than static policies. Trace-driven simulation also assists in examining each policy in detail. Results showed that the proposed data clustering and adaptive cleaning significantly reduced 20.8~21.6% of erase operations and 85.1~88.4% of blocks copied. Flash memory was more evenly worn as well.

RELATED WORK

Many storage systems adopt the non-in-place-update scheme that requires garbage collection to reclaim obsolete space. The Log-Structured File System (LFS)[14-16] is a representative. Since garbage collection in LFS is similar to flash memory cleaning, several cleaning policies proposed in LFS have been employed in flash memory storage systems.[5] The *greedy* policy selects the segment with the largest amount of garbage for cleaning, and was shown to perform well for uniform accesses but poorly for high localities of reference.[6,11,12,14,15] The *cost-benefit* policy considers not only the amount of garbage but also the age of data, and was shown to outperform greedy policy for high localities of reference.[11,12,14,15] The *age sorting* method,[14,15] which sorts data blocks by age before writing them on disks, is used to separate hot data from cold data. Several segments are cleaned at once.

HP AutoRAID,[17] a two-level disk array structure, uses the *hole plugging* method in garbage collection that reclaims a segment by overwriting its valid data to other segments' *holes* (space occupied by obsolete data). Matthews et al.[18] recently proposed an *adaptive cleaning* policy that incorporates the hole plugging into traditional LFS cleaning. This policy adapts to changes in disk utilization by dynamically choosing cost-benefit policy or hole-plugging policy.

Logical Disk (LD),[19,20] using the structure for maintaining a logical block map in a LFS-like file system, also provides a data clustering method. It supports the abstraction of *block lists* to allow file systems to express the logical relationships among blocks. Segment cleaner then can use the list information to physically cluster related blocks to improve future read performance during cleaning. Its clustering effectiveness depends on the correct specifications of block lists.

Douglis et al.[4] discussed storage alternatives for mobile computers. They showed that the key to flash memory file system is erasure management and found that flash memory utilization has substantial impact. For 90% utilization, energy consumption is increased by 70~190%, write response time is degraded by 30%, and lifetime is decreased by up to a third, as compared to 40% utilization. In addition, at 90% utilization or above, an erasure unit much larger than the file system block size will result in many unnecessary copying.

Microsoft's Flash File System (MFFS)[13] employs the greedy policy in cleaning, and so does the Linux PCMCIA[21] flash memory drivers.[22,23] The Linux's approaches sometimes choose to clean the segment that has been erased the fewest number of times is the only difference. The idea is to avoid all erasures concentrated on a few segments.

eNVy[6], a large flash memory-based storage system, provides flash memory as linear

memory array rather than emulated disks. eNVy uses hardware support of copy-on-write and page-remapping techniques to provide transparent in-place update semantic. The *hybrid cleaning* policy combines *FIFO* and *locality gathering* to minimize the cleaning costs for uniform access and high localities of reference. The *flash cleaning cost* was devised as a metric to evaluate cleaning policies. The flash memory utilization is the only factor in the flash cleaning cost.

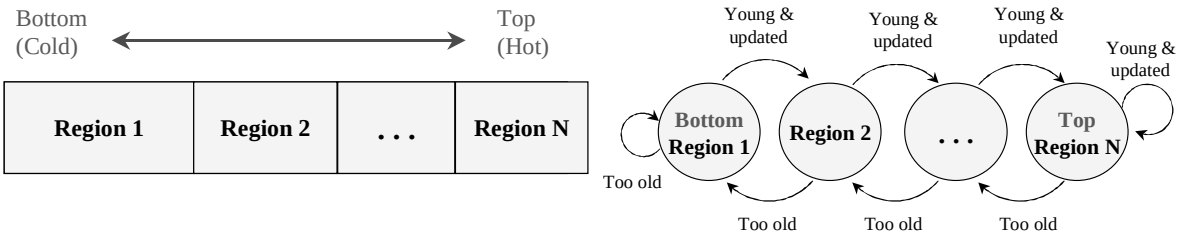
Kawaguchi et al.[5] designed a flash memory file system for UNIX based on LFS. They modified LFS's cost-benefit policy but used different cost measure. The *separate segment cleaning* method is used in data clustering, which uses two segments for cleaning: one for cleaning cold segments and one for writing the data and cleaning the not-cold segments. Separate segment cleaning was shown to perform better than only one segment is used for both the data writes and the cleaning operations,[5,12] since hot data are less likely to be mixed with cold data. Wear leveling is not implemented in their work.

In our early study, the *CAT* policy[11] is proposed to take into account utilization, segment age, and the number of times segments have been erased in selecting segments to clean. Because the number of erase operations performed on individual segments is concerned, flash memory is more evenly worn than greedy policy and cost-benefit policy. Valid blocks in the segments to be cleaned are migrated into separate segments depending on whether the blocks are hot or cold. CAT policy and cost-benefit policy were shown to outperform greedy policy for high localities of reference but do not perform as well as greedy policy for uniform access.[11,12] Data reorganization method was shown to be the most important factor in affecting cleaning performance.[12]

FLASH MEMORY MANAGEMENT USING DATA CLUSTERING

Data reorganization by separating hot data from cold data can reduce cleaning overhead. Previous research[5,6,11,12] reorganizes data only at the cleaning time when valid data in the segment to be cleaned are migrated. We noted that data reorganization can not only be done during cleaning, but also can be done during data updating time without extra cost. By taking advantage of the chance that when data are updated, they are updated to another free flash space, hot data and cold data can be separately clustered.

We propose a new data reorganization method to dynamically cluster data: **DAC** (**D**ynamic



(a) Partitioning flash memory into regions. (b) State transition diagram.

Figure 1. State machine for data clustering

d**A**ta **C**lustering) approach. The approach logically partitions the flash memory into several *regions*, as shown in Figure 1(a). Data blocks in the same regions have similar write access frequencies. Only write operations are concerned since read operations do not incur cleaning. As their access frequencies change over time, data blocks are actively migrated between regions; that is, data blocks are moved toward the *Top* region (i.e. the hottest) if their update frequencies increase, and are moved toward the *Bottom* region (i.e. the coldest) if their update frequencies decrease. By this active data migration, data with similar access frequencies can be effectively clustered.

A state machine is used for the region switching. The state machine contains several states and the state transition diagram is shown in Figure 1(b). Each data block is associated with a state that indicates the region it resides in. The starting state is "*Bottom region*," in which data blocks reside when they are newly created. The state switching occurs only when data blocks are updated or when garbage collection is needed.

A data block is "promoted" to the upper region toward the Top when it is updated; that is, the obsolete data block in the original region is invalidated as garbage and the update data are written to free space in the upper region. Otherwise, the update data are written to the free space in the current region. A block that is valid is "demoted" to the lower region toward the Bottom when the segment it belongs to is selected for cleaning; that is, the block is migrated back to the lower region by being copied into free space in the lower region. Otherwise, the block is migrated to the free space in the current region.

Because the hot degree of a block should not consider only the number of times it is updated, the age of data should also be taken into account. We thus add an additional criterion for state switching into the state machine: the time threshold. If a block is to be promoted, it also has to be young to the current region (i.e. the resident time in the current region is smaller than a certain threshold). If a block is to be demoted, it also has to be old to the current region (i.e., its resident time exceeds a certain threshold).

By this active data migration between neighboring regions during data updating time and during cleaning time, *Top* region will gather the most frequently updated data during the recent accesses. The closer to the Top region, the hotter the block is; otherwise, the colder it is. Therefore, data blocks of similar write access frequencies can be effectively clustered. Figure 2 shows the detailed operations.

The advantages of DAC method can be summarized as follows. First, data are clustered in a light-overhead way during data updating and during segment cleaning. So complex computations for determining data as hot or cold are not needed as in previous research.[5,11,12] Second, data classification is more fine-grained. Instead of classifying data into hot and cold as in previous research,[5,11,12] the DAC approach is more fine-grained since more states of data are allowed depending on the configuration of the state machine. Since data reorganization by separating hot data from cold data can reduce cleaning overhead, this fine-grained classification is expected to perform better than coarse-grained classification in reducing cleaning overhead. Table I lists the comparison of various data-clustering methods.

Table I. Comparison of various data-clustering methods

	Age Sorting	Separate Segment Cleaning (Segment-based)	Separate Segment Cleaning (Block-based)	Dynamic Data Clustering
Clustering method	Sort by age	Classify into hot and cold	Classify into hot and cold	Classify by update frequency
Clustering time	Cleaning	Cleaning	Cleaning	Cleaning and update time
Storage system	LFS[14,15]	Flash memory-based file system[5]	Flash memory server[11,12]	DAC server

```
Write()
{
    If newly write {
        Allocate a free block in Bottom region;
        Write data into the free block;
    } Otherwise
        /* non-in-place update */
        Mark the obsolete data as invalid;
        If the data block is young to the current region
            Allocate a free block in the upper region;
        Otherwise
            Allocate a free block in the current region;
        Write data into the free block;
    }
}

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

Figure 2. Algorithms for write and cleaning

ADAPTIVE CLEANING POLICY

Previous studies[5,6,11,12,14,15,18,24] showed that the performance of static cleaning policies are sensitive to data access behavior and that no single cleaning policy performs well for all access patterns in selecting segments to clean. For example, cost-benefit policy[5,14,15] and CAT policy[11,12] perform well for locality of references whereas greedy policy[5,6,11,12,14,15] performs well for uniform access. Since data access patterns may change over time, an *adaptive cleaning* policy is proposed. Because read accesses do not incur cleaning, only write accesses are considered.

The adaptive cleaning policy combines CAT policy and greedy policy to react to changes in the data access patterns by dynamically choosing policies. When the recent references exhibit uniform access, greedy policy is used; otherwise, CAT policy is used. Therefore, the problem of adaptive cleaning becomes how the cleaner knows the recent write access patterns.

An approximate method is devised to determine whether uniform access has occurred. The idea is to use a *Write Monitor* to monitor the incoming write requests. The Write Monitor maintains a *block reference table* to count the number of times each block has been accessed. When cleaning is needed, an *Analyzer* examines the numbers in the block reference table. If the variance is very large (by comparing with the variance if uniform access has occurred), then non-uniform access is reported.

After cleaning, all the counters in the block reference table will be reset to zero. The period for monitoring write references begins from the last cleaning time to the current cleaning time. Therefore, only recent accesses affect the reference analysis.

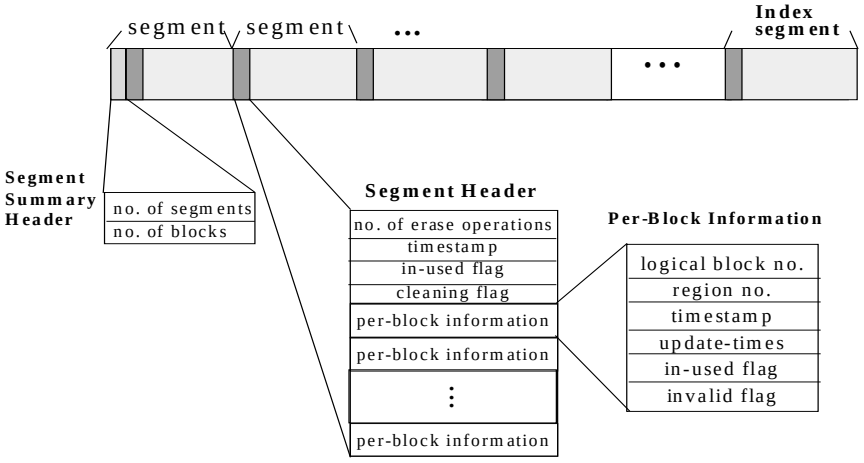


Figure 3. Data layout on flash memory

A flash memory server providing the DAC data clustering and the adaptive cleaning was implemented. The server manages flash memory as fixed-size blocks and uses the *non-in-place-update* scheme for data updating. 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.

We first describe data layout on flash memory, then introduce three tables that the server uses to speed up processing. They are region table, translation table, and lookup table. These tables are constructed in main memory during server startup time by obtaining segment information from flash memory and are maintained during runtime. The information stored in tables is only copies of information stored in flash memory. Therefore, even if power failures occur, these tables can be reconstructed from flash memory.

Data layout on flash memory

Figure 3 shows the data layout on flash memory. Each segment has a *segment header* to record segment information such as the number of times the segment has been erased, timestamp, and *per-block information array*. The per-block information array describes every block in the segment, including logical block number, region number that the block belongs to, timestamp, the number of times the block has been updated, and flags to indicate whether obsolete or not. The *segment summary header* keeps information about flash memory, including total number of flash segments and total number of blocks per segment. The final segment, *index segment*, keeps track of indexes to the actively written segments at the current time.

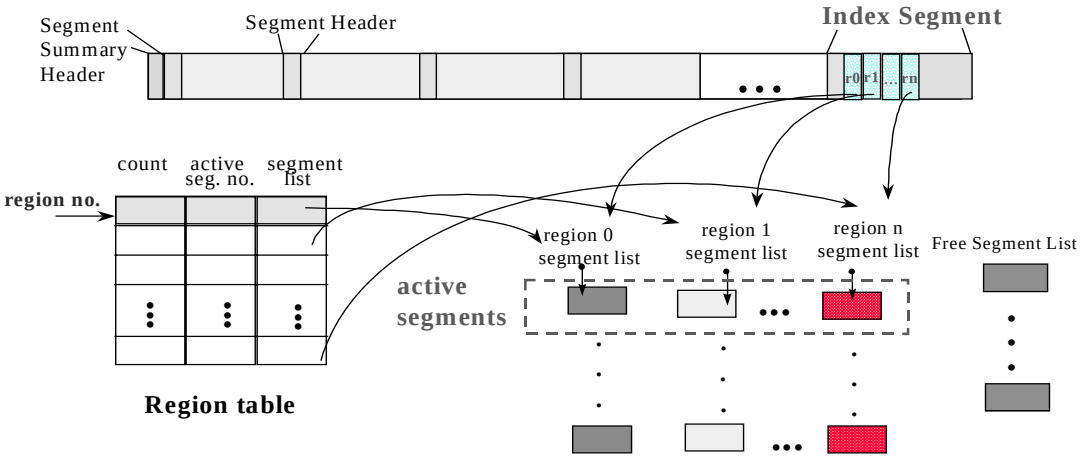


Figure 4. Region table and region segment lists

Region management

Since DAC data clustering is used, flash memory is logically partitioned into several regions. A *region table*, shown in Figure 4, keeps track of information for each region, including the total number of segments in the region, the currently active segment for data writing, and a *region segment list*. The region segment list keeps track of each segment in the region. This table is used by the cleaner in deciding which segments to clean.

The *active segments* record segments that are currently used for data writing in each region and are stored in the index segment on flash memory. A *free segment list* records the available free segments. Initially, the server reads segment headers from flash memory to identify free segments to construct the free segment list at server startup time. When an active segment is out of free space, a segment from the free segment list is used as the active segment and the change of active segment is written to the index segment as an appended log. When there is no free space, the index segment is erased first before wrapping around the log. When the number of free segments is below a certain threshold, the cleaner is activated to reclaim the garbage.

Block-based translation table

Since data blocks are not updated in place, their physical locations in flash memory change when updated. A *translation table*, shown in Figure 5, records the physical locations for each logical block, which is constructed to speed up the address translation from logical block numbers to physical addresses in flash memory. Therefore, every logical block has an entry in the translation table to record its segment number and block number. Each entry also contains a region number indicating which region the block belongs to and a timestamp indicating when this block is allocated in the region. These information together are used by the DAC state machine to decide whether a block's state should be switched and whether data should be migrated between neighboring regions.

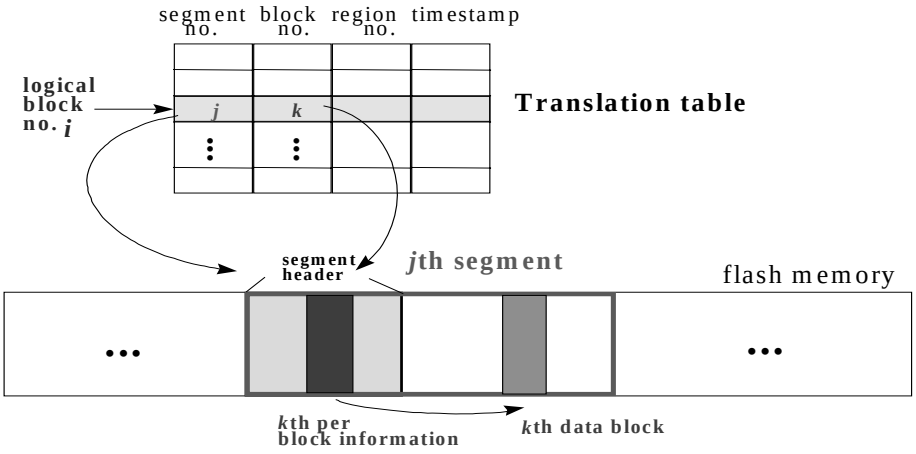


Figure 5. Translation table and address translation

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

Figure 6. Lookup table to speed up cleaning

When a block is updated to another new empty block, the old per-block information is marked *invalid* and the new per-block information records the logical block number. The corresponding translation table entry is also updated to record the current physical location.

Segment-based lookup table

The *lookup table*, shown in Figure 6, records information about each segment such as the number of times the segment has been erased, segment creation time, and control flags for cleaning. The server also does bookkeeping of *valid block count* to count the number of valid blocks in a segment. Cleaner then uses this segment information to speed up the selection of segments for cleaning. Since the table is constructed in main memory, the selection is fast. The table also contains the *first free block* (the first free block available for writing in a segment) to speed up the block allocation.

SIMULATION STUDIES

Trace-driven simulation was performed to examine the cleaning effectiveness of the proposed DAC data reorganization and alternative cleaning policies. The impact of flash memory utilization, flash memory size, and degree of locality of reference were examined in detail as well. To evaluate the cleaning effectiveness of various policies, we first devised several metrics. We then introduce the simulator and traces, finally we present simulation results.

Metrics

To measure the amount of work involved in cleaning, we devised a formula to express flash memory cleaning cost. Since, before a segment is erased and reclaimed, valid data should be migrated by copying to free space in other segments, the flash memory cleaning cost includes erasure cost and migration cost, which can be expressed as the following formula:

$$CleaningCost_{Flash\ Memory} = NumberofErase*(EraseCost+MigrateCost_{valid\ data}) \quad (1)$$

The cost of each erasure is constant, whereas the migration cost is determined by the amount of data migrated during cleaning. The larger the amount of valid data, the higher the migration cost. However, the erasure cost dominates the migration cost. In order to provide better performance and prolong flash memory lifetime, the primary goal is to minimize the number of erase operations performed, then to minimize the number of blocks copied during cleaning.

Because the time to write a segment is about certain times of the time to erase a segment, if we express the ratio of write time to erase time as *WritetoEraseTimeRatio*, then formula (1) can be re-formulated as

Table II. Simulator parameters

Flash size	Flash memory size.
Flash segment size	The size of an erasure unit.
Flash segment lifecycle	Program/erase cycles.
Flash block size	The block size that the server maintains.
Flash storage utilization	The amount of initial data, relative to flash size. Data are preallocated in flash memory at the start of simulation.
Number of regions	Number of regions (i.e. the states of the DAC state machine).
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.

$$CleaningCost_{Flash\ Memory} = NumberofErase * EraseCost + \frac{TotalNumberofBlocksCopied}{NumberofBlocksPerSegment} * WritetoEraseTimeRatio * EraseCost$$

$$= EraseCost * (NumberofErase + \frac{TotalNumberofBlocksCopied}{NumberofBlocksPerSegment} * WritetoEraseTimeRatio) \quad (2)$$

Since erasure cost is constant, we then use the following simplified formula derived from formula (2) as the metric to compare the cleaning effectiveness of various cleaning policies:

$$SimplifiedCleaningCost_{Flash\ Memory} = NumberofErase + (\frac{TotalNumberofBlocksCopied}{NumberofBlocksPerSegment} * WritetoEraseTimeRatio) \quad (3)$$

Since another important goal for flash memory storage systems is wear leveling, the *degree of uneven wearing*[12] which indicates the variance of wearing for flash segments is also used as the other metric. A utility was created to read the number of erase operations performed on individual segments from flash memory. The standard deviation of these numbers is computed as the degree of uneven wearing. The smaller the standard deviation, the more evenly the flash memory is worn.

Simulator

Our flash simulator completely simulated the flash memory server except that it stores data in a large memory array instead of flash memory. The simulator accepts the parameters as shown in Table II and provides the following four segment selection algorithms:

- greedy policy (**Greedy**)

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

- cost-benefit policy[5] (**Cost-benefit)**

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

- CAT policy[12] (**CAT)**

  The cleaner chooses to clean segments that minimize the formula: $\frac{u}{(1-u)*a} * t$, where u is utilization, a is segment age, and t is the number of times the segment has been erased.

- adaptive cleaning policy (**Adaptive)**
  When uniform access is detected, greedy policy is used; otherwise, CAT policy is used.

For simplicity, the simulator assumes each request can be finished before arrival of next request. The number of erase operations performed on segments, the number of blocks copied, the simplified cleaning cost, and the degree of uneven wearing are reported for each simulation.

Traces

Ruemmler and Wilkes[25,26] have collected the disk-level traces of HP-UX workstations at Hewlett-Packard Laboratories. The traces covering two-month activities are collected on three different systems: a time-sharing computer, a file server, and a personal workstation. The personal workstation (**hplajw**) was used mainly for e-mail and document editing. Since the usage behaviors of personal computers are likely to be similar to what would be used on mobile computers, hplajw traces were often used in simulations of mobile computers,[4,12,27-29] where flash memory products are typically applied. So our simulations used the hplajw traces to drive the simulator.

The traces original exhibit high locality of references and 71.2% of writes were to metadata.[26] Since flash memory capacity is currently still small, we do not expect flash memory to contain swap space, so traces from swap partition were excluded. This exclusion is not supposed to have much impact on the correctness of results since swap partition occupies only 7.1% of the writes in the traces.[26] Totally, 1331-Mbyte references are writes.

Simulation Results for hplajw Traces

Though flash memory capacity is currently still small, the capacity is increasing as hardware technologies advance. So we simulated the flash memory as large as the hard disk used in the hplajw (i.e. 278 Mbytes). Because of the need of extra space for segment headers and cleaning, the simulated flash memory was 283 Mbytes with 128-Kbyte erase segments. The server maintains data in 1-Kbyte blocks. Because the typical time to write a segment is 0.4~0.6 seconds and to erase a segment is 0.6~0.8 seconds, we used 0.75 as the ratio of write time to erase time in calculating the cleaning cost.

The effect of DAC data clustering

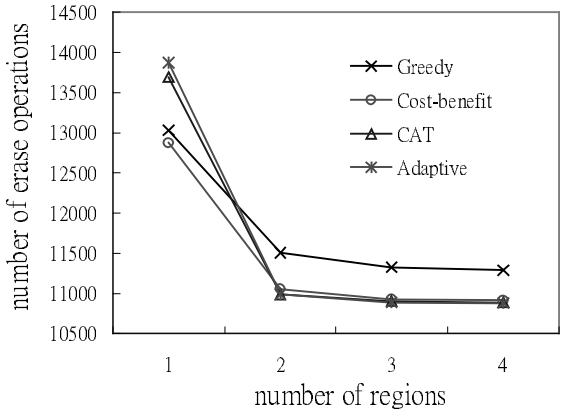
We first measured the effect of DAC data clustering for various cleaning policies. The number of regions (i.e., the number of states in the DAC state machine) ranged from 1 to 4. Because the time threshold for state switching will affect the performance, to fairly compare the effectiveness of various cleaning policies, the time threshold was not set. That is, a block's state is promoted when it is updated and demoted when the segment it belongs to is selected for cleaning. The effect of time threshold was measured in the next simulation.

Because cleaning activities were low under low utilization, in order to measure the cleaning overhead, flash memory utilization was initially set to 85% by writing enough blocks in sequence to fill the flash memory to 85% of flash memory space, then hplajw traces were used. Figure 7 shows the results. As shown in Figure 7(a)-(c), when DAC data clustering was used (i.e. the number of regions is more than 1), each policy reduced large amounts of erase operations and blocks copied; cleaning cost was thus greatly reduced. For example, Adaptive

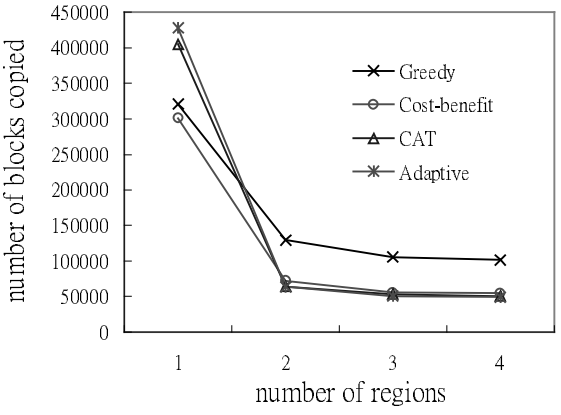
incurred 20.8~21.6% fewer number of erase operations, 85.1~88.4% fewer number of blocks copied, and 30.8-32% less cleaning cost. When DAC state machine was configured with 4 states (i.e. 4 regions), each policy reached the best performance and Adaptive performed best.

Figure 7(d) shows that CAT performed best in the wear leveling, since CAT formula takes even wearing into account in selecting segments to clean. The degree of uneven wearing for Adaptive is only slightly worse than CAT since Adaptive is a hybrid policy of CAT and Greedy. Flash memory was worn unevenly for Greedy and Cost-benefit, because they do not consider even wearing when selecting segments to clean.

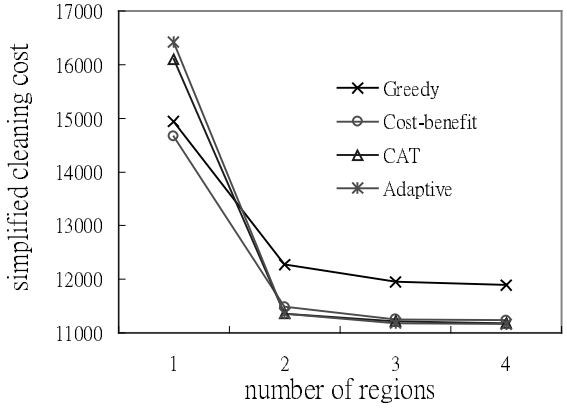
Figure 7(e) shows the breakdown of policy selections in Adaptive. 75~79% of times, the adaptive cleaner chose CAT policy because locality of access was reported. This is close to the fact that 71.2% of writes in traces are to metadata. Though we used 0.75 as the ratio of write time to erase time, Figure 7(f) shows the impact of ratios on the simplified cleaning costs: the larger the ratio is, performance gaps among cleaning policies widen.



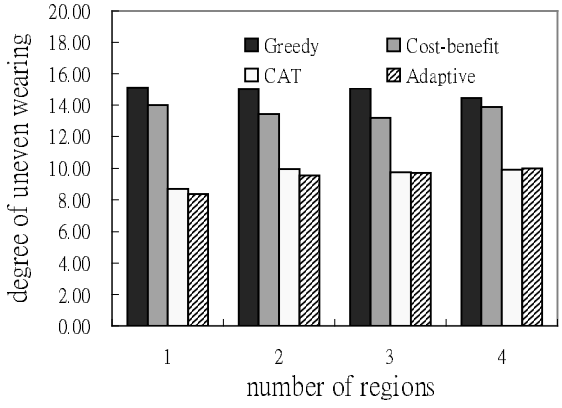
(a) Number of erase operations.



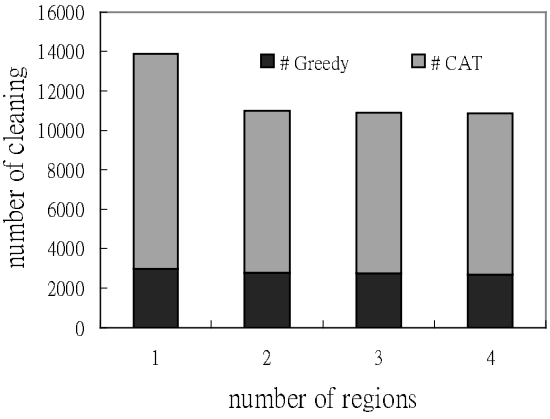
(b) Number of blocks copied during cleaning.



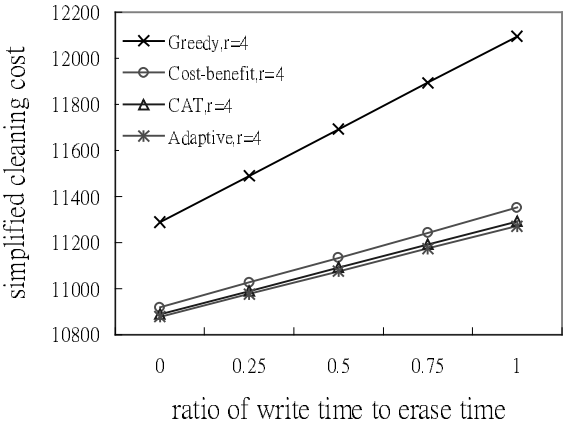
(c) Cleaning cost



(d) Degree of uneven wearing.



(e) Breakdown of policy selections in adaptive cleaning.



(f) Varying the ratio of write time to erase time.

Figure 7. Effect of DAC data clustering for the DAC state machine configured with different number of states

Varying the time threshold for state switching

The *time threshold for state switching* is an additional criterion to control whether a block's state must be promoted when the block is updated or must be demoted when the segment the block belongs to is selected for cleaning. Figure 8 shows the results of varying the time threshold. We found that under this workload, for a 4-state DAC state machine, not to set the time threshold (i.e. labeled '*' in the x-axis) is better than to set it. However, in our experiences with the other workloads (e.g., workloads in the simulations for varying localities of reference), setting the threshold can further reduce cleaning overhead. Therefore, whether setting the time threshold for region transition improves performance depends largely on workloads and data access behaviors.

Varying flash memory utilization

Since our DAC data clustering significantly improved performance under high utilization, we wanted to find out how performance varies under various degrees of utilization. Therefore, before each run of simulation, we wrote enough blocks in sequence to fill the flash memory to the desired level of utilization, then used hplajw traces. In this measurement, the DAC state machine was configured with 4 states.

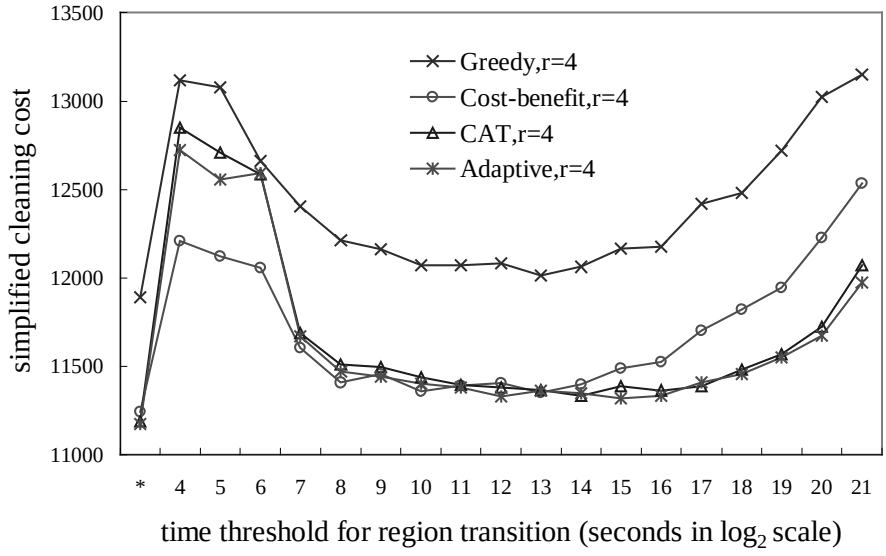
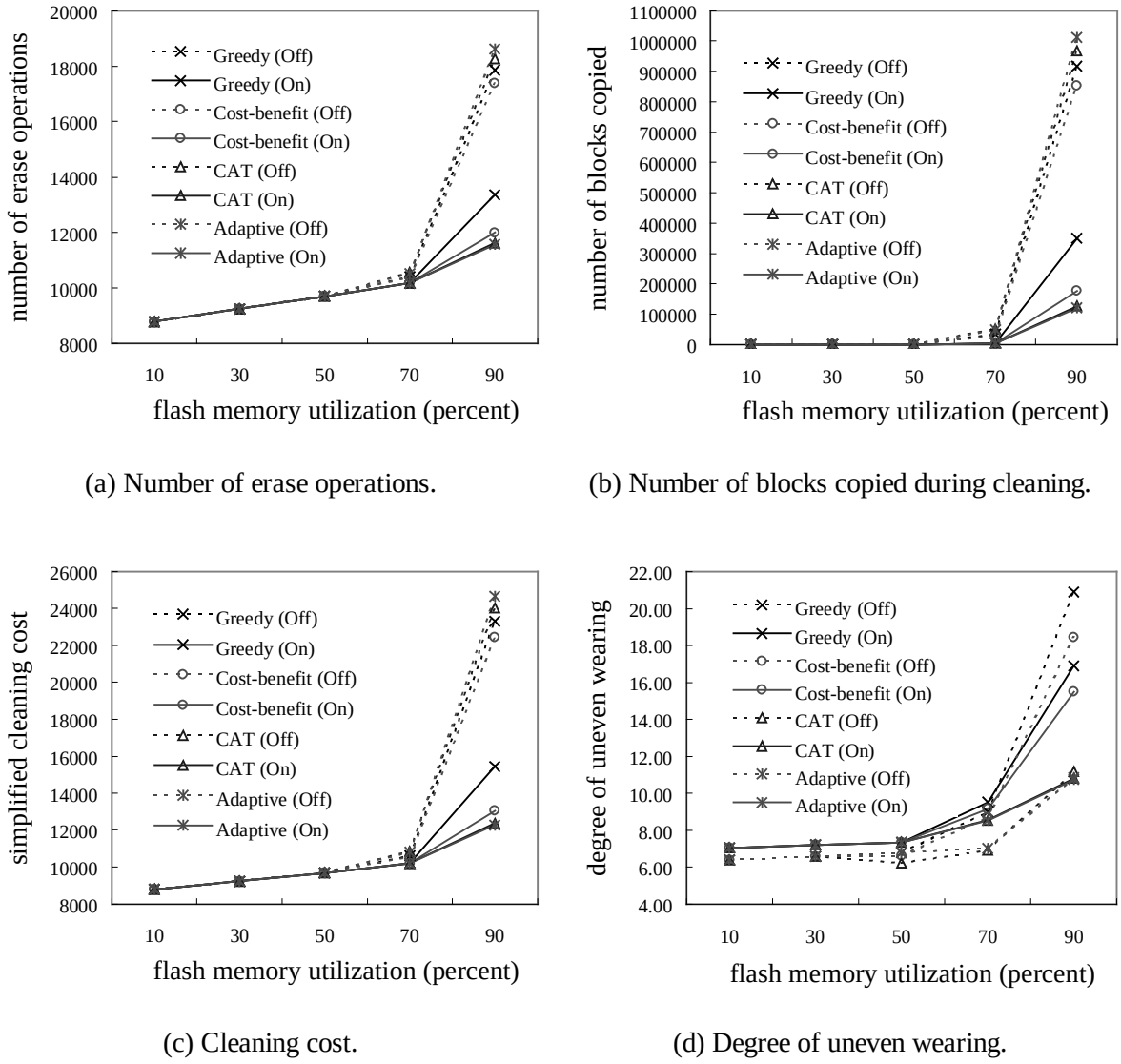


Figure 8. Effect of varying the time threshold for state switching in the DAC state machine

Figure 9 shows that as the utilization increased, performance degraded for each policy. The degradation is because more space was occupied by valid data, and then more cleaning was needed in order to reclaim enough free space. Therefore, the effect of different cleaning methods was especially prominent for higher utilization. At utilization above 70%, each policy degraded dramatically when DAC data clustering was not used, whereas each degraded gradually when DAC data clustering was used. At 90% utilization, the performance gaps of cleaning cost for using or not using DAC were 33.8% for Greedy, 41.8% for Cost-benefit, 48.5% for CAT, and 50.2% for Adaptive.

Varying flash memory size

In order to know the impact of flash memory size on the cleaning, the traces were preprocessed to map to flash memory space before simulation. Figure 10 shows that as the size of flash memory increased, each policy performed better since more free space was left and then



(a) Number of erase operations.

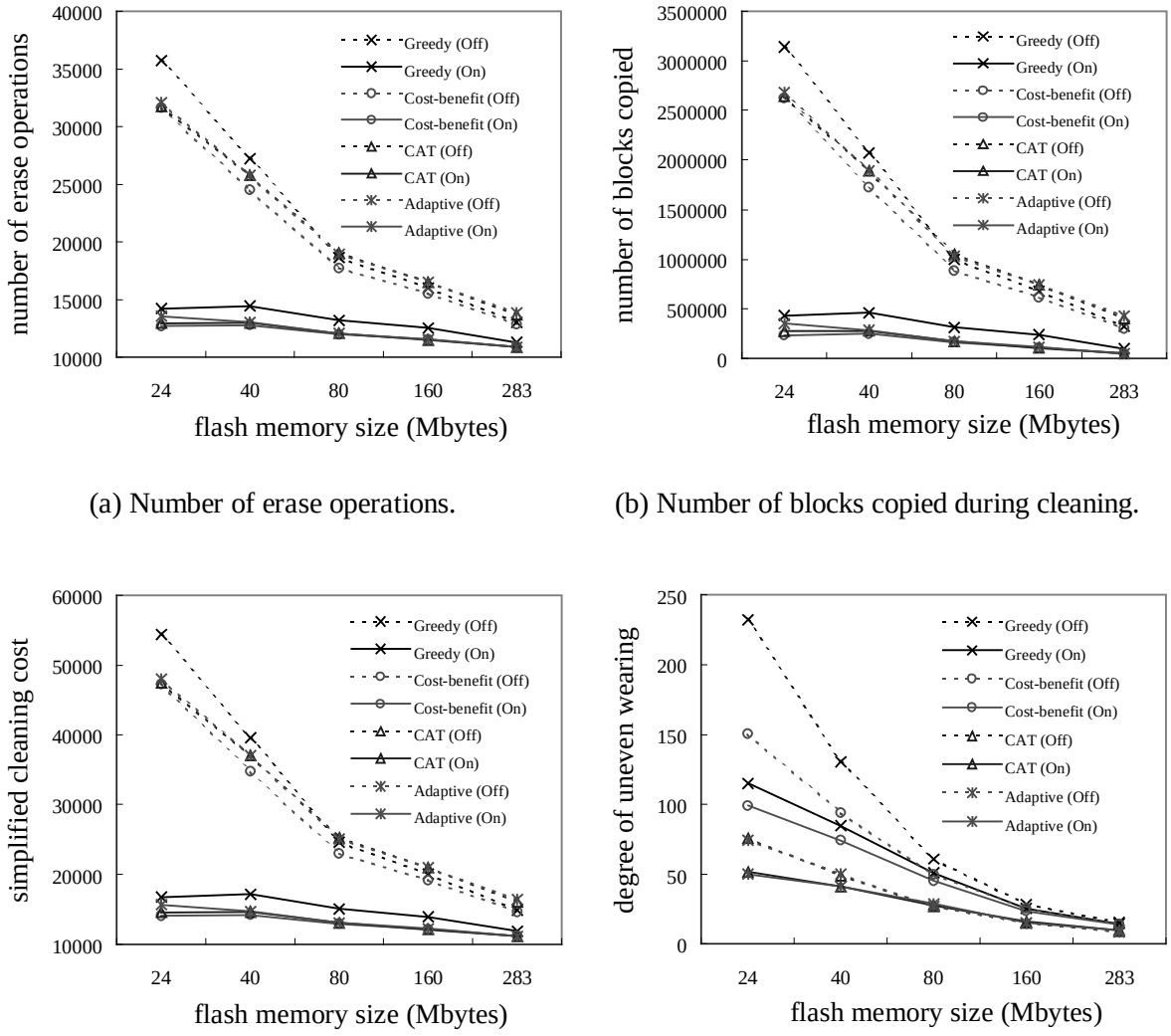
(b) Number of blocks copied during cleaning.

(c) Cleaning cost.

(d) Degree of uneven wearing.

Figure 9. Varying flash memory utilization. Policies used with DAC data clustering were labeled "On" in the legends; otherwise, they were labeled "Off".

cleaning was less needed. However, performance for each policy depended largely on flash memory size when DAC data clustering was not used, whereas each policy performed well for various sizes when DAC data clustering was used (4-state DAC state machine was used in this simulation).



(a) Number of erase operations.

(b) Number of blocks copied during cleaning.

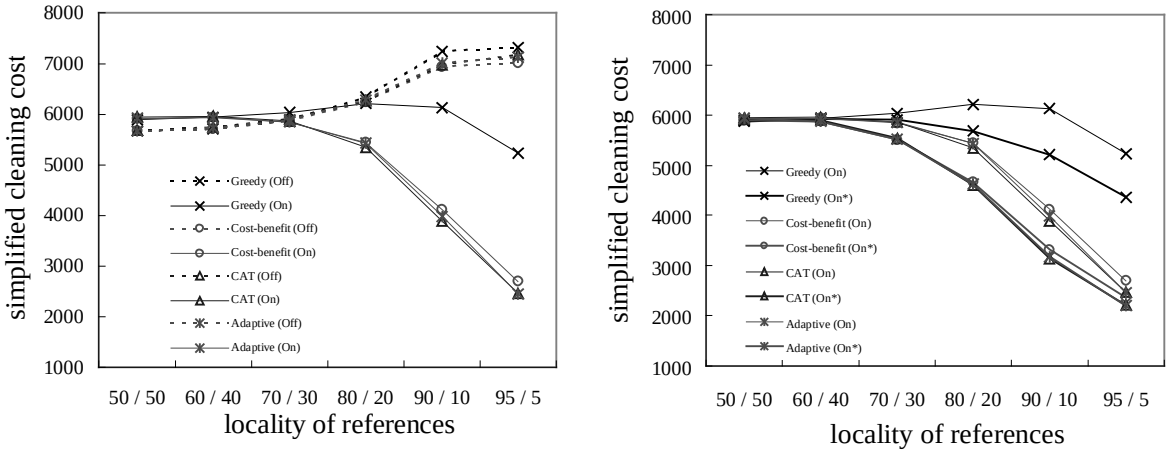
(c) Cleaning cost.

(d) Degree of uneven wearing.

Figure 10. Varying flash memory size. Policies used with DAC data clustering were labeled "On" in the legends; otherwise, they were labeled "Off".

Varying degree of locality

Because HP traces exhibit high localities of reference, we wanted to find out whether DAC data clustering performs well for various localities of reference. A workload generator based on the *hot-and-cold* workload used in the evaluation of Sprite LFS cleaning policies[14,15] was created to generate workloads for different localities of reference. The generated workload was 14-day 192-Mbyte write references in 4-Kbyte units. The inter-arrival rate of requests is



(a) Using DAC data clustering to reduce cleaning cost.

(b) Effect of setting time threshold for state switching.

Figure 11. Varying localities of reference. Policies used with DAC data clustering were labeled "On" in the legends when the time threshold for state switching was not set, were labeled "On" when the time threshold was set; otherwise they were labeled "Off".*

Poisson distribution. The flash memory was 24 Mbytes with 128-Kbyte segments. The block size was 4 Kbytes and flash memory utilization was set to 85% before each run of simulations.

Figure 11(a) shows the results. We used the notation "x/y" for locality of reference, in which x% of all accesses go to y% of the data while (1-x)% goes to the remaining (1-y)% of data. As the locality increased, the cleaning costs for each policy were greatly increased when DAC data clustering was not used, but dramatically decreased when DAC data clustering was used (4-state DAC state machine was used and the time threshold for state switching was not set). The performance gaps of cleaning cost for using or not using DAC were 1.9~28.5% for Greedy, 0.5~61.5% for Cost-benefit, 0.8~65.6% for CAT, and 0.6~65.6% for Adaptive. This shows that DAC data clustering can significantly reduce cleaning costs by effective data clustering. This effect is especially prominent for higher locality of references. Figure 11(b) shows that setting the time threshold for state switching is beneficial for this kind of workloads. When the threshold was set to 84 minutes (arbitrarily chosen), cleaning costs can be further reduced as compared with those when the time threshold was not set: 0.3~12.5% for Greedy, 0.4~12.5% for Cost-benefit, 0.6~12.5% for CAT, and 0.6~11.9% for Adaptive.

PERFORMANCE EVALUATIONS

A flash memory server utilizing DAC data clustering was implemented on Linux in GNU C++. Table III summarizes the experimental environment. In order to measure the effectiveness of alternate cleaning policies, four policies were also implemented in the server: greedy policy (**Greedy**), cost-benefit policy[5] (**Cost-benefit**), CAT policy[11,12] (**CAT**), and adaptive cleaning policy (**Adaptive**).

Table III. Experimental environment

Hardware:
PC: Pentium 133 MHz, 32 Mbytes of RAM
PC Card Interface Controller: Omega Micro 82C365G
Flash memory: Intel Series 2+ 24Mbyte Flash Memory Card [9,10] (segment size:128 Kbytes)
Software:
Operating system: Linux Slackware 96 (kernel version: 2.0.0, PCMCIA package version: 2.9.5)

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

The block size that the server managed is 4-Kbyte blocks. The number of states that DAC state machine was configured ranged from 1 to 4 and the time threshold for state switching was set to 30 minutes. To initialize the flash memory, enough blocks were written in sequence to fill the flash memory to 90% of flash memory space. Benchmarks were created to overwrite the initial data according to the synthetic workload.

Figure 12 shows that applying DAC data clustering (i.e., the number of regions is more than 1) is beneficial for each policy. Large amounts of erase operations and blocks copied were reduced and the average throughputs were largely increased as well. For example, the number of erase operations for Adaptive were reduced by 15.97~22.55%, the amount of blocks copied were reduced by 20.12~28.41%, the cleaning costs were reduced by 16.5~23.3%, and the throughput improvement was 19.54~25.19%, as shown in Figure 12(a)~(d).

Using DAC data clustering can substantially reduce cleaning overhead, which can be further reduced when an effective cleaning policy is used. Figure 12(c) shows that Adaptive had the lowest cleaning cost. When the number of regions is 4, Adaptive outperformed Greedy by 15.73%, Cost-benefit by 4.67%, and CAT by 0.67%. The advantage over CAT is not prominent, which can be explained as follows. The behavior of Adaptive is like Greedy for random access but like CAT for locality access. Because half the synthetic workloads were random accesses in which Greedy slightly outperformed CAT, Adaptive slightly outperformed CAT. Therefore, we found that effective data clustering is the most important factor in affecting flash cleaning.

Though Adaptive performed best in the reduction of erase operations and blocks copied, it performed 0.1~2% worse than CAT in the average throughput, as shown in Figure 12(d). This is because Adaptive needs the cost of monitoring each write reference and adapts to the changes of write access patterns. As the performance gap between CPU and erase operation widens, we expect Adaptive will outperform CAT since Adaptive incurs fewer amounts of

erase operations and blocks copied than CAT.

Figure 12(e) shows that flash memory was more evenly worn for CAT and Adaptive. Figure 12(f) shows the breakdown of policy selections in adaptive cleaning policy. On an average of 41% of the times the adaptive cleaner used Greedy and used CAT the remaining 59% of the times. Since half of the synthetic workload is random access and the other half is locality access, the 41% of the times to choose Greedy was very close to what we would expect.

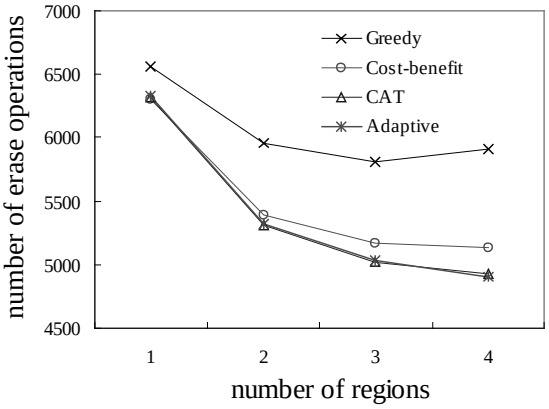
The above results demonstrated that using DAC data clustering which effectively clusters data according to write access frequencies can reduce cleaning overheads and that the adaptive cleaning policy is able to adapt to the change of access patterns.

Discussions for space and time overheads

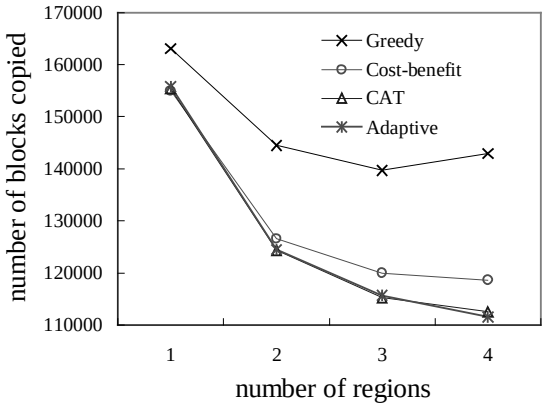
Computations and space overhead are the disadvantages of Adaptive. To adapt to the variations of write access patterns, the adaptive cleaner needs to record each write reference and compute the variance. These computations take up a substantial amount of time. However, Adaptive still performed well in the average throughput since it reduced the largest number of erase operations and blocks copied. The computation overhead is offset by this reduction. As the speed gap between CPU and erase operation widens, we expect the overhead of computation time to be largely reduced.

The block reference table maintained by the adaptive cleaner requires a substantial amount of main memory: 4 bytes per blocks. For example, for a 24-Mbyte flash memory with 4-Kbyte blocks, the table requires 23-Kbyte main memory. Because current flash memory capacity is still small, the space overhead is limited. However, if flash memory capacity becomes large, a small table with certain replacement policy may be appropriate.

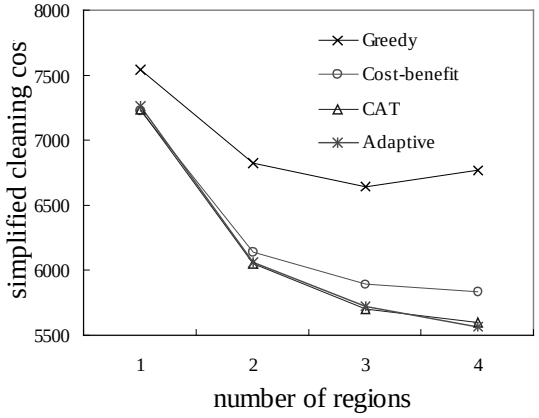
In our implementation of server, the region table, translation table, and lookup table are constructed in main memory. Storing these tables requires a substantial amount of main memory: 12 bytes per region, 13 bytes per block, and 17 bytes per segment. However, it is a trade-off between space consumption and performance. Because currently flash memory capacity is still small, the space overhead is limited. With a 24-Mbyte flash memory, 128-Kbyte segments, 4K-byte blocks, and 4 regions, these tables take up 78 Kbytes of main memory.



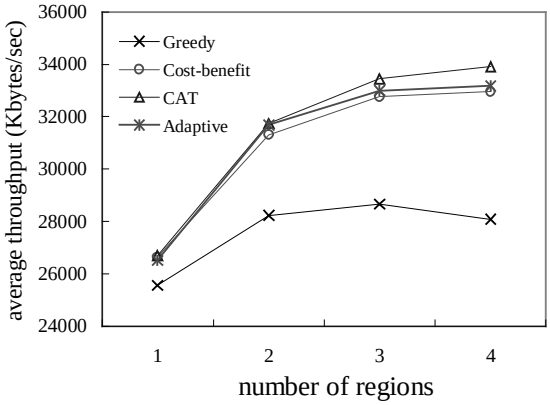
(a) Number of erase operations.



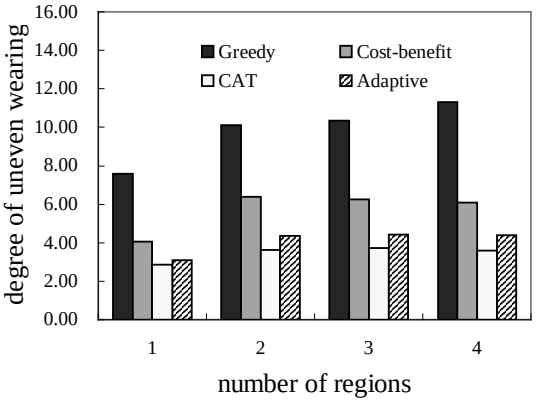
(b) Number of blocks copied during cleaning.



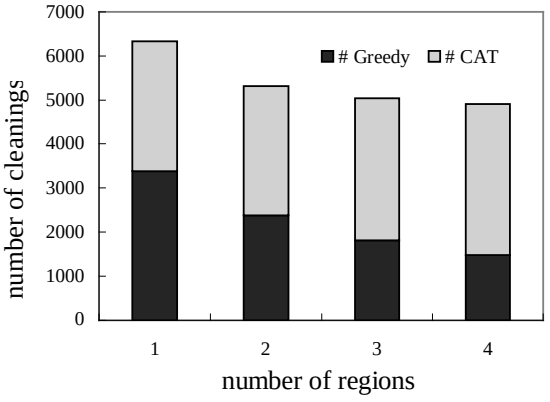
(c) Cleaning cost.



(d) Degree of uneven wearing.



(e) Average throughput.



(f) Breakdown of policy selections in adaptive cleaning.

Figure 12. Performance results of DAC data clustering for synthetic workload

CONCLUSION

Flash memory is expected to be largely used as its capacity increases and the price decreases. Large erases and writes will be created and wear leveling will be very important. Effective cleaning policies help to maximize the flash memory lifetime, improve system performance, and reduce power consumption. In this paper, we have presented a data reorganization technique for clustering frequently accessed data to reduce cleaning overhead. Data are clustered dynamically according to their write access frequencies. An adaptive cleaning policy is also proposed to adapt to the changes of workload and data access patterns.

We have detailed the implementation of a flash memory server utilizing the proposed data clustering technique and adaptive cleaning policy. Performance evaluations with synthetic workloads showed that cleaning cost was significantly reduced by 16.5~23.3% and the throughput improvement was 19.54~25.19%. Trace-driven simulations with real-world traces showed that cleaning cost was reduced by 30.8~32%. Flash memory lifetime is thus extended, system throughput is largely improved, and flash memory is evenly worn. In examining the degree of wear-leveling and exploring the impacts of flash memory utilization, flash memory size, and degree of locality of reference, the proposed methods all performed well.

Several factors are important in determining how well the DAC data clustering will work in a given environment, such as the configuration for the number of states in the DAC state machine and the setting of the time threshold for state switching. In our experience, these factors are highly dependent on workloads. We also noticed that different segment selection algorithms perform differently for the same setting of factors. Besides, how large a difference can be used to judge uniform or non-uniform accesses will affect the behavior of adaptive cleaner.

The proposed methods can not only be used in flash memory, they can also be used in other applications that can benefit from data clustering or need segment cleaning. For example, our evaluations and simulations showed that the number of blocks copied during cleaning was also significantly reduced. The results suggest that applying the proposed data reorganization technique and adaptive cleaning is also beneficial to segment cleaning in the log-based disk systems which have no erasure cost and consider only reducing the number of blocks copied to improve cleaning performance. The other promising application is applying the DAC data clustering to cluster hot disk data together near the center of disk to reduce seek times in disk storage systems. Clustering frequently accessed data can reduce seek times and improve disk performance has been shown in many researches.[30-35]

ACKNOWLEDGMENTS

We would like to thank John Wilkes at Hewlett Packard Laboratories, who graciously made his HP I/O traces available for our simulations and provided valuable comments. We would also like to thank David Hinds for his valuable comments and help on our work.

REFERENCES

1. M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, ‘Non-Volatile Memory for Fast, Reliable File Systems’, *Proceedings of the 5th International Conference on*

Architectural Support for Programming Languages and Operating Systems, pp. 10-22, Boston, MA, October 1992.

2. R. Caceres, F. Douglis, K. Li, and B. Marsh, 'Operating System Implications of Solid-State Mobile Computers', *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pp. 21-27, Napa, CA, October 14-15, 1993.
3. B. Dipert and M. Levy, *Designing with Flash Memory*, Annabooks, 1993.
4. F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, 'Storage Alternatives for Mobile Computers', *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 25-37, 1994.
5. A. Kawaguchi, S. Nishioka, and H. Motoda, 'A Flash-Memory Based File System', *Proceedings of the 1995 USENIX Technical Conference*, pp. 155-164, New Orleans, LA, January 16-20, 1995.
6. M. Wu and W. Zwaenepoel, 'eNVy: A Non-Volatile, Main Memory Storage System', Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 86-97, San Jose, CA, October 1994.
7. N. Ballard, 'State of PDAs and Other Pen-Based Systems', *Pen Computing Magazine*, Aug. 1994, pp. 14-19.
8. T. R. Halfhill, 'PDAs Arrive But Aren't Quite Here Yet', *BYTE*, Vol. 18, No. 11, 1993, pp. 66-86.
9. Intel, *Flash Memory, 1994.*
10. Intel Corp., 'Series 2+ Flash Memory Card Family Datasheet', http://www.intel.com/design/flcard/datashts, 1997.
11. M. L. Chiang, Paul C. H. Lee, and R. C. Chang, 'Managing Flash Memory in Personal Communication Devices', *Proceedings of the 1997 International Symposium on Consumer Electronics (ISCE'97)*, pp. 177-182, Singapore, December 2-4, 1997.
12. M. L. Chiang and R. C. Chang, 'Cleaning Policies in Mobile Computers Using Flash Memory', Submitted to *Journal of Systems and Software*.
13. P. Torelli, 'The Microsoft Flash File System', *Dr. Dobb's Journal*, pp. 62-72, February 1995.
14. M. Rosenblum, 'The Design and Implementation of a Log-Structured File System', *PhD Thesis*, University of California, Berkeley, June 1992.
15. M. Rosenblum and J. K. Ousterhout, 'The Design and Implementation of a Log-Structured File System', *ACM Transactions on Computer Systems*, 10, (1), 26-52, (February 1992).
16. M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, 'An Implementation of a Log-Structured File System for UNIX', *Proceedings of the 1993 Winter USENIX Conference*, pp. 307-326, January 1993.
17. J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, 'The HP AutoRAID Hierarchical

Storage System', *ACM Transactions on Computer Systems*, 14, (1), 108-136, (February 1996).

18. J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson, 'Improving the Performance of Log-Structured File Systems with Adaptive Methods', *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, Saint Malo, France, October 5-8, 1997.
19. W. de Jonge, M. F. Kaashoek, and W. C. Hsieh, 'Logical Disk: A Simple New Approach to Improving File System Performance', Technical Report MIT/LCS/TR-566, Massachusetts Institute of Technology, 1993.
20. W. de Jonge, M. F. Kaashoek, and W. C. Hsieh, 'The Logical Disk: A New Approach to Improving File Systems', *Proceedings of 14th Symposium on Operating Systems Principles*, pp. 15-28, 1993.
21. D. Anderson, *PCMCIA System Architecture*, MindShare, Inc. Addison-Wesley Publishing Company, 1995.
22. D. Hinds, 'Linux PCMCIA HOWTO', http://hyper.stanford.edu/~dhinds/pcmcia/doc/PCMCIA-HOWTO.html, v2.5, February 19, 1998.
23. D. Hinds, 'Linux PCMCIA Programmer's Guide', http://hyper.stanford.edu/~dhinds/pcmcia/doc/PCMCIA-PROG.html, v1.38, February 4, 1998.
24. T. Blackwell, J. Harris, M. Seltzer,'Heuristic Cleaning Algorithms in Log-Structured File Systems', *Proceedings of the 1995 USENIX Technical Conference*, pp. 277-288, New Orleans, LA, January 16-20, 1995.
25. C. Ruemmler and J. Wilkes, 'A Trace-Driven Analysis of Disk Working Set Sizes', *Technical Report HPL-OSR-93-23*, Hewlett-Packard Laboratories, Palo Alto, CA, April 5, 1993.
26. C. Ruemmler and J. Wilkes, 'UNIX Disk Access Patterns', *Proceedings of the 1993 Winter USENIX*, pp. 405-420, San Diego, CA, January 25-29, 1993.
27. K. Li, 'Towards a low power file system', Technical Report UCB/CSD 94/814, University of California, Berkeley, CA, May 1994, Masters Thesis.
28. K. Li, R. Kumpf, P. Horton, and T. Anderson, 'A Quantitative Analysis of Disk Drive Power Management in Portable Computers', *Proceedings of the 1994 Winter USENIX*, pp. 279-291, San Francisco, CA, 1994.
29. B. Marsh, F. Douglis, and P. Krishnan, 'Flash Memory File Caching for Mobile Computers', *Proceedings of the 27 Hawaii International Conference on System Sciences*, pp. 451-460, Maui, HI, 1994.
30. S. Akyurek and K. Salem, 'Adaptive Block Rearrangement', *ACM Transactions on Computer Systems*, 13, (2), 89-121 (1995).
31. S. Akyurek and K. Salem, 'Adaptive Block Rearrangement Under UNIX', *Software-Practice and Experience*, 27, (1), 1-23, (January 1997).

32. C. Ruemmler and J. Wilkes, 'Disk Shuffling', *Technical Report HPL-91-156*, Hewlett-Packard Laboratories, Palo Alto, CA, October 28, 1991.

33. C. Staelin and H. Garcia-Molina, 'Clustering Active Disk Data to Improve Disk Performance', *Technical Report CS-TR-283-90*, Department of Computer Science, Princeton University, September 1990.

34. C. Staelin and H. Garcia-Molina, 'Smart Filesystems', *Proceedings of the 1991 Winter USENIX*, pp. 45-51, Dallas, TX, 1991.

35. P. Vongsathorn and S. D. Carson, 'A System for Adaptive Disk Rearrangement', *Software-Practice and Experience*, 20, (3), 225-242, (1990).