A Flash-Memory Based File System

Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda
Advanced Research laboratory, Hitachi, Ltd.

Abstract

A flash memory device driver that supports a conventional UNIX file system transparently was designed. To avoid the limitations due to flash memory's restricted number of write cycles and its inability to be overwritten, this driver writes data to the flash memory system sequentially as a Log-structured File System (LFS) does and uses a cleaner to collect valid data blocks and reclaim invalid ones by erasing the corresponding flash memory regions. Measurements showed that the overhead of the cleaner has little effect on the performance of the prototype when utilization is low but that the effect becomes critical as the utilization gets higher, reducing the random write throughput from 222 Kbytes/s at 30% utilization to 40 Kbytes/s at 90% utilization. The performance of the prototype in the Andrew Benchmark test is roughly equivalent to that of the 4.4BSD Pageable Memory based File System (MFS).

1. Introduction

Flash memory, a nonvolatile memory IC (Integrated Circuit) that can hold data without power being supplied, is usually a ROM (Read Only Memory) but its content is electrically erasable and rewritable. The term “flash” is used to indicate that it is a whole chip or a block of contiguous data bytes (We call this block an *erase sector*). Many kinds of flash memory products [1][2] are available, and their characteristics are summarized in Table 1†.

Because flash memory is five to ten times as expensive per megabyte as hard disk drive (HDD) memory, it is not likely to become the main mass storage device in computers. Its light weight, low energy consumption, and shock resistance, however, make it very attractive for mass storage in portable computers. In fact, flash memory in the form of an IC-card commonly replaces the HDD or is used for auxiliary storage in portable personal computers.

Read Cycle	80 - 150 ns
Write Cycle	10 µs/byte
Erase Cycle	1 s/block
Cycles limit	100 000 times
Sector size	64 Kbytes
Power Consumption	30 - 50 mA in an active state 20 - 100 µA in a standby state
Price	10 - 30 $/MByte

Table 1. Flash memory characteristics.

Flash memory has two other disadvantages limiting its use in computer systems. One is that its content cannot be overwritten: it must be erased before new data can be stored. Unfortunately, this erase operation usually takes about one second. The other disadvantage is that the number of erase operations for a memory cell is limited, and upper limits on the order of 100 000 erasures are not unusual. An advantage of flash memory, however, is that its read speed is much greater than that of a HDD. The performance of flash memory in read operations is, in fact, almost equivalent to that of conventional DRAM.

Our objective in the work described here was to explore the possibilities of using flash memory in file systems and to develop an experimental but practical flash memory based file system for UNIX. We used a log approach to ensure that new data was always written in a known location—so that the erase operation could be performed in advance.

2. Design and Implementation

We have designed and implemented a flash memory device driver that emulates a HDD and supports a standard UNIX file system transparently. We chose the device driver approach for its simplicity.

† There is another type of flash memory that has much smaller erase sectors. (See Section 4.)

Since flash memory can be accessed directly through a processor's memory bus, other approaches (such as tightly coupling a flash memory system and a buffer cache) might perform better by reducing memory-to-memory copy operations. Such an approach, however, would require a large number of kernel modification because flash memory's erase and write properties differ greatly from those of the main memory.

2.1 Overview

Our driver must record which regions contain invalid data and reclaim the invalid region by erasing those regions. Furthermore, since the number of erase operations in each region is limited, the driver must at least monitor the number in order to assure reliable operation. In some cases, wear-leveling should be provided.

Our driver maintains a sequential data structure similar to that of LFS [3][4]. It handles a write request by appending the requested data to the tail of the structure. To enable later retrieval of the data it maintains a translation table for translating between physical block number and flash memory address. Because the translation is made on the level of the physical block number, our driver can be viewed, from the file-system aspect, as an ordinal block device.

When write operations take place the flash memory system is fragmented into valid and invalid data blocks. To reclaim invalid data blocks, we use a cleaner that selects an erase sector, collects valid data blocks in the sector, copies their contents to the tail of the log, invalidates the copied blocks (and consequently makes the entire sector invalid), and issues an erase command to reclaim the sector. The functions of this cleaner are identical to those of LFS's cleaner. Our prototype does not implement wear-leveling, although it does maintain a log of each erased sector.

2.2 Flash Memory Capability

Early generations of flash memory products prohibit read or write operations while they are performing a write or an erase operation. That is, when such a flash memory chip is performing an erase operation on an erase sector, data can neither be read from nor written to other erase sectors until the erase operation completes. A naive file system implementation ignoring this prohibition would not be feasible in a multitasking environment because it would unpredictably block operations whenever the program wanted data from a flash memory chip that

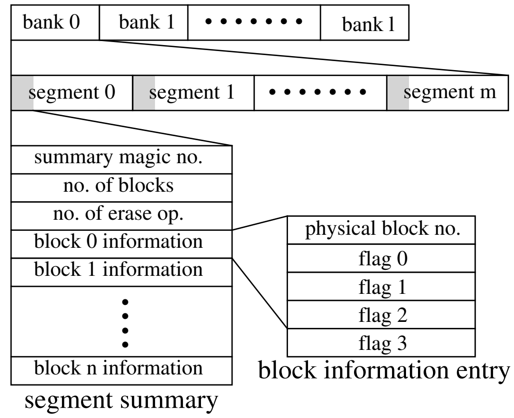

Figure 1. On-chip data structure.

was performing an erase operation triggered by another program. This problem can be avoided by temporarily caching in a buffer all valid data in the flash memory chip to be erased. This caching operation, however, could consume a significant amount of processor resources.

Recent flash memory products provide an erase-suspend capability that enables sectors that are not being erased to be read from during an erase operation. Some new products also support write operations during the erase suspension. Our driver assumes the underlying flash memory system to be capable of erase- suspended read operations.

Flash memory generally takes more time for a write operation than for a read operation. It provides a write bandwidth of about 100 Kbytes/s per flash memory chip, whereas a conventional SCSI HDD provides a peak write bandwidth 10 to 100 times higher. Some recent flash memory products incorporate page buffers for write operations. These buffers enable a processor to send block data to a flash memory chip faster. After sending the data, the processor issues a “Page Buffer Write” command to the chip and the chip performs write operations while the processor does other jobs.

Our driver assumes that the underlying flash memory system consists of some banks of memory that support concurrent write operations on each chip. This assumption reduces the need for an on-chip page buffer because the concurrent write operations can provide a higher transfer rate.

2.3 On Flash Memory Data Structure

Figure 1 depicts our driver's data structure built on an underlying flash memory system. The flash memory system is logically handled as a collection of banks. A bank corresponds to a set of flash memory chips and each set can perform erase or write operations independently. The banks are in turn divided into segments, each of which corresponds to an erase sector of the flash memory system. Each segment consists of a segment summary and an array of data blocks. The segment summary contains segment information and an array of block information entries. Segment information includes the number of blocks the segment contains and the number of times the segment has been erased.

2.4 Flag Update

Each block information entry contains flags and the physical block number to which this data block corresponds. The physical block number is provided to the driver by the file-system module when issuing a write data request. The flags are written sequentially so that the driver can record the change of the block status without erasing the segment.

The driver uses four flags to minimize the possibility of inconsistency and to make recovery easier. When a logical block is overwritten the driver invalidates the old block, allocates a new block, and writes new data to the newly allocated block. The driver updates the flags on the flash memory in the following order:

- **Step 1.** mark the newly allocated block as `allocated`,
- **Step 2.** write the block number and then write new data to the allocated block,
- **Step 3.** mark the allocated block as `pre-valid`,
- **Step 4.** mark the invalidated block as `invalid`, and
- **Step 5.** mark the allocated block as `valid`.

The above steps guarantee that the flag values of the newly allocated and invalidated blocks never become the same under any circumstances. Therefore, even after a crash (e.g., a power failure) during any one of the above steps, the driver can choose one of the blocks that hold the fully written data. This method is of course not sufficient to maintain complete file system consistency, but it helps suppress unnecessary ambiguity at the device level.

2.5 Bank Management

To manage block allocation and cleaning, the driver maintains a bank list and a cleaning bank list. Figure 2 shows the relationship between these lists. The driver allocates a new data block from the active bank, so data write operations take place only on the active bank. When the free blocks in the active bank are exhausted, the driver selects from the bank list the bank that has the most free segments (i.e., free blocks) and makes it the new active bank.

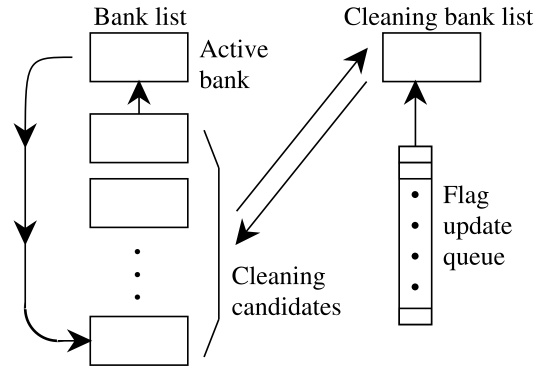

Figure 2. Bank list and cleaning bank list.

When a segment is selected to be cleaned, the bank containing that segment is moved to the cleaning bank list. The bank stays in the list until an erasure operation on the segment finishes. Because the bank is no longer on the bank list, it never becomes an active bank, and thus avoids being written during the erase operation.

The driver maintains the flag update queue to handle the flag update procedure, described in the previous section, on blocks in the bank of which segment is being erased. The driver avoids issuing a data write on a bank being cleaned by separating the bank list and the cleaning bank list. However, when a block is logically overwritten, an invalidated block might belong to that bank. In such a case, the driver postpones the flag update procedure steps 4 and 5, by entering the pair of the newly allocated and the invalidated blocks into the queue. All the pairs are processed when the erasure finishes. Note that even if the pairs are not processed due to a crash during the erasure, the driver can recover flag consistency because of the flag update order (Step 3 for each pair has been completed before the crash occurs.).

The queue should be able to hold the number of pairs that are expected to be entered during an erasure. For example, the current implementation can generate 500 pairs for 1 erasure [i.e., 500 blocks (250 KBytes) per second], and thus has 600 entries in the queue. Should the queue be exhausted, the driver will stop writing until the erasure is complete. We have not yet experienced this condition.

2.6 Translation Table

The translation table data structure contains all information needed to manage the translation of a block number to an address and to manage the erase log of each segment. During the system boot time, the driver scans the flash memory system and constructs this translation table and other structures from the on-chip segment summaries.

Figure 3 shows the relationship between the translation table and the block information entries. During the system boot time the driver scans all the segment summaries one by one. If it finds a valid block, it records a triplet `(bank no., segment no., block no.)` describing the block in a table entry indexed by the physical block number.

After the boot, the driver refers only to the translation table to access data blocks on the flash memory when a read operation is requested. The address of each block can be computed from the triplet. The driver translates a requested physical block number to the address of a corresponding flash memory data block and simply copies the contents of the data block to the upper layer.

When a write operation is requested, the driver checks whether it has already allocated a flash memory data block for a requested physical block. If it has, the allocated block is invalidated. The driver allocates a new flash memory data block, updates the translation table, and copies the requested data to the newly allocated block, while updating the flags.

2.7 Cleaner

The segment cleaning operation takes place during the allocation process when the number of available flash memory blocks for writing becomes low. This operation selects a segment to be cleaned, copies all valid data in the segment to another segment, and issues a block erase command for the selected segment. The cleaning process is the same as that of LFS except that it explicitly invokes the erase operation on the segment.

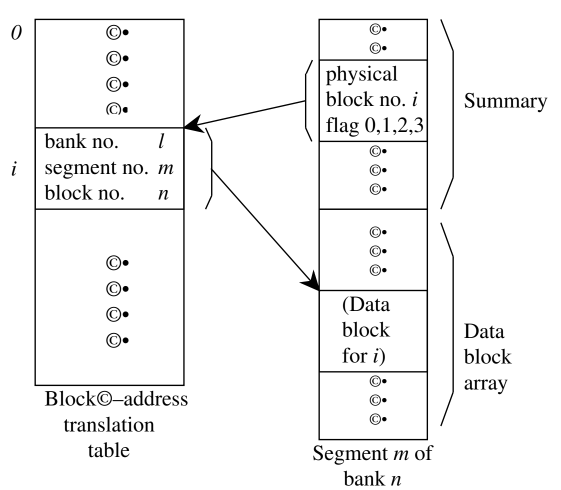

Figure 3. Relationship between the block–address translation table and a block information entry.

Command	Description
`FLIOCCWAIT`	Wait until a segment is selected to be cleaned.
`FLIOCCCBLK`	Copy 16 valid blocks of the selected segment. Return 0 if no more valid blocks exist in the segment .
`FLIOCCIERS`	Start erasing the segment. Return 0 when the erasure is complete.

Table 2.`ioctl` commands for the cleaner.

The cleaner is divided into three parts: policy, copying and erasing, and internal data maintenance. All jobs are executed in kernel-space, though copying and erasing are conducted by a daemon running in user-space.

As discussed in [4], implementing a cleaner as a user process makes the system flexible when changing or adding a cleaning policy or algorithm, or both. By communicating with the kernel through the use of system calls and the *ifile,* the BSD LFS cleaner does almost all jobs in user-space. Our driver, in contrast, does the cleaning jobs in kernel-space as Sprite LFS does. We use a daemon to make the copying process concurrent with other processes. We took this approach for its ease of implementation.

While data is being written, cleaning policy codes are executed when a block is invalidated. If cleaning policy conditions are satisfied for a segment, the driver adds it to the cleaning list and wakes up the cleaner daemon to start copying valid blocks. Upon awakening, the cleaner daemon invokes the *copy* command repeatedly until all valid blocks are copied to the clean segments. Then, it invokes the *erase* command and the driver starts erasing the segment by issuing an erase command of the flash memory. The copying is performed by codes within the driver. The cleaning daemon controls the start of the copying. It makes the copying concurrent with other processes.

We added three `ioctl` commands for the cleaner daemon (Table 2). The daemon first invokes `FLIOCCWAIT` and then waits (usually) until a segment to be cleaned is selected. As an application program writes or updates data in the file system, the device driver eventually encounters a segment that needs to be cleaned. The driver then wakes up the cleaner daemon and continues its execution. Eventually, the daemon starts running and invokes `FLIOCCCBLK` repeatedly until all the valid blocks are copied to a new segment. On finishing the copy operation, the daemon invokes `FLIOCCIERS`, which causes the driver to issue an erase command to the flash memory. The daemon invokes `FLIOCCWAIT` again and waits until another segment needs to be cleaned.

2.8 Cleaning Policy

For our driver, the cleaning policy concerns:

- When the cleaner executes, and,
- Which segments is to be cleaned.

The flash memory hardware permits multiple segments to be erased simultaneously as long as each segment belongs to the different bank. This simultaneous erasure provides a higher block-reclaim rate. For simplicity, however, the current implementation cleans one segment at a time. The cleaner never tries to enhance logical block locality during its copying activity. It simply collects and copies live data in a segment being cleaned to a free segment.

In order to select a segment to clean, the driver is equipped with two policies: "greedy" and "cost-benefit" [3] polices. The driver provides `ioctl` commands to choose the policy. The greedy policy selects the segment containing the least amount of valid data, and the cost-benefit policy chooses the most valuable segment according to the formula:

$$\frac{\text{benefit}}{\text{cost}} = \frac{age \times (1-u)}{2u},$$

where u is the utilization of the segment and *age* is the time since the most recent modification (i.e., the last block invalidation). The terms $2u$ and $1\text{-}u$ respectively represent the cost for copying (u to read valid blocks in the segment and u to write back them) and the free space reclaimed. Note that LFS uses $1+u$ for the copying cost because it reads the whole segment in order to read valid blocks in the segment for cleaning.

The cleaning threshold defines when the cleaner starts running. From the point of view of the load put upon the cleaner, the cleaning should be delayed as long as possible. The delay should produce more blocks to be invalidated and consequently reduce the number of valid blocks that must be copied during the cleaning activity. Delaying the cleaning activity too much, however, reduces the chances of the cleaning being done in parallel with other processes. This reduction may markedly slow the system.

Our driver uses a gradually falling threshold value that decreases as the number of free segments decreases. The curve A in Figure 4 shows the threshold of the current implementation. It shows that

- When enough (N) free segments are available, the cleaner is not executed,
- When the number of free segments becomes smaller than N and if there are some segments whose policy accounts are greater than T_h, cleaning starts with the segment that has the greatest policy accounts, and
- As the number of free segments becomes smaller, the threshold becomes lower so that more segments can be chosen with lower policy accounts.

For the greedy policy of the current implementation, N is 12 and T_h is 455 invalid blocks. That is, when the number of free segments becomes 12, segments that contain more than 455 invalid blocks are cleaned. For the cost-benefit policy, N is 12 and T_h is set to the value that is equivalent to being unmodified 30 days with one invalid block. For both the policies, segments having no valid blocks are always cleaned before other segments.

The threshold curve enables the driver to stop the cleaner as long as enough free segments are available and also to start the cleaner at a slow pace. For example, suppose the driver employs a policy such as "When the number of free segments becomes N_b, start the cleaner." (This policy is represented by the threshold curve B in Figure 4.) When the number of free segments became N_b the cleaner would start cleaning even if the most invalidated segment had only one invalid block. Furthermore, if the live data in the file system counted more than N_s-N_b segments (where N_s is a total number of segments), the cleaner would run every time a block was invalidated. This would result in a file system that was impractically slow and had an impractically short lifetime.

3. Performance Measurements and Discussion

Unlike a HDD-based file system, the prototype is free from seek latency and it is thu expected to show nearly the same performance for both sequential and random read operations. In fact, for reading 4Kbyte blocks from 12.6 Mbytes of data, the sequential and random throughputs of the driver are respectively 644 and 707 Kbytes/s. (For the same tasks, the through throughputs of MFS [7] are 647 and 702 Kbytes/s.)

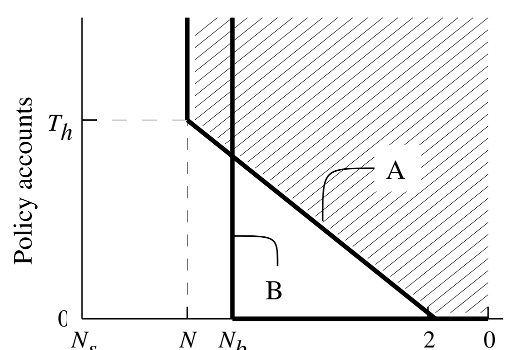

Figure 4. Cleaning threshold.

Flash Memory System	
Flash memory	Intel 28F008SA (1 Mbyte/chip)
Banks	8
Segments	64 (8 segments/bank)
Segment size	256 Kbytes
Data blocks	32256 (504 blocks/segment)
Erase cycle	1 sec.
bandwidth	252 Kbytes/sec.
Write bandwidth	400 Kbytes/sec.
Read bandwidth	4 Mbytes/sec.
CPU and Main Memory System	
CPU	IDT R3081 (R3000 compatible)
Cache	Instruction 16 Kbytes
	Data 4 Kbytes
Memory bandwidth	
Instruction read	20 Mbytes/sec.
Data read	7 Mbytes/sec.
Data write	5 Mbytes/sec.

Table 3. Test-platform specifications.

The write performance of the prototype, on the other hand, is affected by the cleaning, as is the case with LFS.

The benchmark platform consists of our hand-made computer running 4.4BSD UNIX with a 40MHz R3081 processor and 64 Mbytes of main memory. The size of the buffer cache is 6 Mbytes. Table 3 summarizes the platform specification†.

3.1 Sequential Write Performance

The goal of our sequential write performance test was to measure the maximum throughput that can be expected under certain conditions. When a large amount of data is written sequentially, our driver invalidates blocks in each segment sequentially. The driver therefore needs no copying for cleaning a segment and the maximum write performance can be obtained.

Figure 5 shows the sequential write throughput as a function of cumulative data written, and Table 4 summarizes the results. The results were obtained by first writing a certain amount of data and then repeatedly overwriting that initial data. The curves show results based on different initial data: 4.2 Mbytes (30% of the file system capacity), 8.4 Mbytes (60%), and 12.6 Mbytes (90%). The greedy policy was used for cleaning.

The results obtained with the 90% initial data load were unexpected. Although the data were overwritten sequentially, many blocks were copied for cleaning. This copying was a result of the effect of the cleaning threshold described earlier. Since the live data counts

† The actual hardware had 128 segments (16 segments/block), but in the work reported here we used only half the segments of each bank.

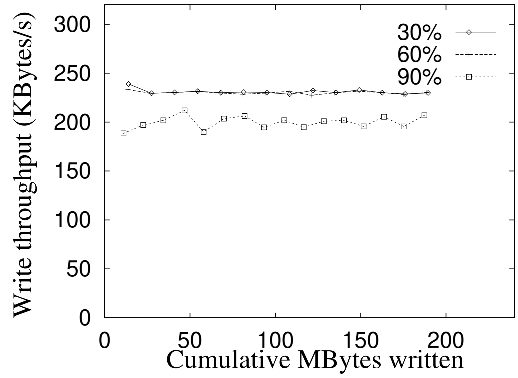

Figure 5. Sequential write performance.

Initial data	Average throughput (Kbytes/s)	Number of erased segments	Number of copied blocks	Total data written (Mbytes)
30%	231	749	0	192
60%	230	766	0	192
90%	199	889	57 266	192

Table 4. Summary of sequential write performance.

more than 53 segments for the 90% data, the cleaning threshold was kept near 410 invalid blocks in a segment throughout the test. Consequently, the cleaner copied an average of 64 blocks per erasure and lowered the write throughput.

3.2 Random Write Performance

The random write performance test evaluated the worst case for our driver. When a randomly selected portion of a large amount of data is overwritten, all the segments are invalidated equally. If the invalidation takes place unevenly (e.g., sequentially), some segments are heavily invalidated, and thus can be cleaned with a small amount of copying. The even invalidation caused by the random update, however, results in there being less chance to clean segments that are particularly highly invalidated. Therefore, the cleaning cost approaches a constant value for all segments.

For our driver, the copying cost is expected to be a function of the ratio of used space to free space in the file system. As new data are written to the free segments, the used segments are invalidated evenly. The free segments are eventually exhausted and the cleaner starts cleaning. Consequently, the ratio of valid to invalid blocks of each segment becomes that of the ratio of used to free space of the file system.

Figure 6 and Table 5 show the results of the random write test. These results were obtained by writing a 4Kbyte data block to a randomly selected position of various amounts of initial data: again, 4.2,

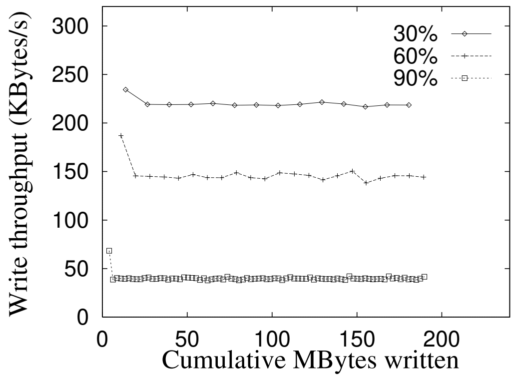

Figure 6. Random write performance.

Initial data	Average throughput (Kbytes/s)	Number of erased segments	Number of copied blocks	Total data written (Mbytes)
30%	222	801	26 383	192
60%	147	1066	155 856	192
90%	40	2634	938 294	192

Table 5. Summary of random write performance.

8.4, and 12.6 Mbytes. And again the greedy policy was used for cleaning.

3.3 Hot-and-Cold Write Performance

This test evaluated the performance cases where write accesses exhibited certain amounts of the locality of reference. In such cases, we can expect the cost-benefit policy to force the segments into a bimodal distribution where most of the segments are nearly full of valid blocks and a few are nearly full of invalid blocks [3]. Consequently, the cost-benefit policy would result in a low copying overhead. We will see that our first results did not match our expectations; we will then analyze why this anomaly occurred and what measures we took to address it.

Figure 7 and Table 6 show the results of the test. 640-116 means that 60% of the write accesses go to one-eighth of the initial data and other 40% go to another one-eighth. The other three-fourths of the data are left unmodified. With this distribution we intend to simulate meta data write activity on an actual file system. The ratio of writes is based on the results reported in [5], which found that 67-78% of writes are to meta data blocks. In all the tests we conducted, an actual write position in a selected portion of the initial data was decided randomly, the size of the initial data was 12.6 Mbytes (90%), and all writes were done in 4Kbyte units.

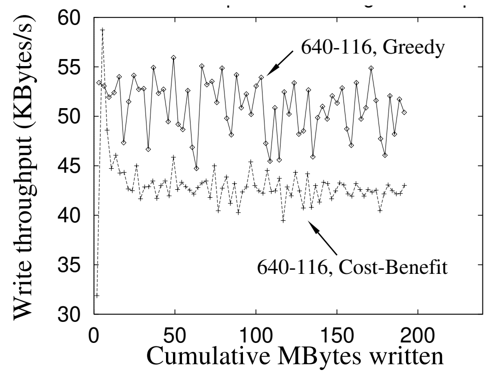

Figure 7. Hot-and-cold write performance.

Test	Cleaning policy	Average throughput (Kbytes/s)	Nomuber of erased segments	Nomuber of copied blocks
640-116	Greedy	51	2617	925 193
	Cost-Benefit	43	3085	1 161 090

Initial data: 90% (12.6 Mbytes), Total data written: 192 Mbytes

Table 6. Summary of hot-and-cold write performance.

3.4 Separate Segment for Cleaning

The initial results obtained in the hot-and-cold write test were far worse than we had expected. The write throughput of the 640-116 test was nearly the same as that of the random test using the greedy policy. Furthermore, the greedy policy worked better than the cost-benefit policy for the 640-116 test.

Figure 8 shows the distribution of segment utilization after the 640-116 test. In the figure, we can observe a weak bimodal distribution of segment utilization. Since the 60% of the data was left unmodified, more fully valid segments should be present.

We traced the blocks that the cost-benefit policy once judged as cold in the test, and Figure 9 shows the distribution of the cold and the not-cold blocks in the segments after executing the 640-116 test. The data in

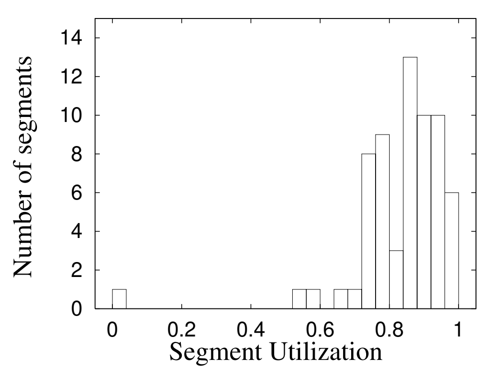

Figure 8. Segment utilization distribution after the 640-116 test.

Initial data	Separate segment	Average throughput (Kbytes/s)	Number of erased segments	Number of copied blocks
30%	No	241	742	0
	Yes	239	744	0
60%	No	197	888	63 627
	Yes	198	832	33 997
70%	No	135	1195	214 894
	Yes	143	883	57 205
80%	No	81	1855	544 027
	Yes	127	1089	157 922
90%	No	43	3085	1 161 090
	Yes	60	2218	723 582

Total data written: 192 Mbytes

Table 7. Summary of 640-116 tests using the separate cleaning segment.

this figure was obtained by marking a block as "cold" when the segment to which the block belongs was chosen to be cleaned and its utilization was less than the average utilization in the file system. We can see that some segments contain both cold and not-cold blocks. Furthermore, the number of cold blocks is much smaller than expected: since three-fourths of the 12.6 Mbytes of initial data were left unmodified, we would expect, in the best case, about 19 000 cold blocks (i.e., about 38 cold segments). In the test, however, the actual number of cold blocks was 2579.

The reason we determined for the above results is that the driver uses one segment for both the data writes and the cleaning operations; the valuable, potentially cold blocks are mixed with data being written to the segment. The number of cold blocks therefore does not increase over time.

To address this problem, we modified the driver so that the driver uses two segments: one for cleaning cold segments and one for writing the data and cleaning the not-cold segments. Table 7 summarizes the results of 640-116 tests on both the modified and the original drivers. The effect of the separate cleaning segment becomes notable as the initial utilization grows, and the write throughput was improved more than 40% for the 90% initial data. Figure 10 shows the distribution of cold blocks after the 640-116 test using the modified driver. Many cold segments are observed.

		MFS	Prototype 52-56%	Prototype 92-96%
Average elapsed time for each run	Phase 1	1.3	2.0	2.9
	Phase 2	8.0	9.5	11.5
	Phase 3	13.1	13.5	13.5
	Phase 4	16.9	16.9	17.1
	Phase 5	80.6	81.8	84.0
	Total	119.9	123.7	129.0
Number of written blocks for data			251 818	233 702
Number of copied blocks			75 227	255 020
Number of erased segments			578	903

Table 8. Andrew Benchmark results.

3.5 Andrew Benchmark

Table 8 lists the results of the Andrew benchmark [6] for MFS and for our prototype. The results were obtained by repeating the benchmark run 60 times. The output data files and directories of each run were stored in a directory, and to limit the file system usage the oldest directory was removed before each run after 14 contiguous runs for the 52-56% test, after 24 for the 92-96% test, and after 9 for the MFS test. Note that, as pointed out in [4], phases 3 and 4 performed no I/O because all the data access were cached by the higher-level buffer and the *inode* caches.

The benchmark consists of many read operations and leaves a total of only about 560 Kbytes of data for each run. As a result, there are many chances to clean segments without disturbing data write operations. Therefore, our prototype shows performance nearly equivalent to that of MFS. We expect that similar access patterns often appear in a personal computing environment. Note that the cleaner erased 903

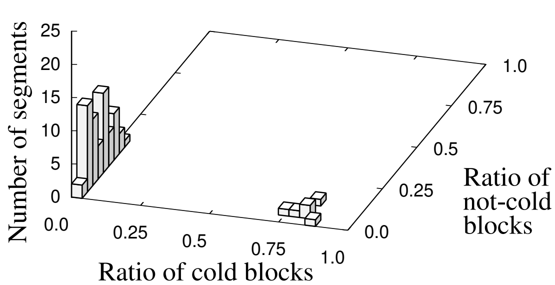

Figure 9. Distribution of cold and not-cold blocks after the 640-116 test.

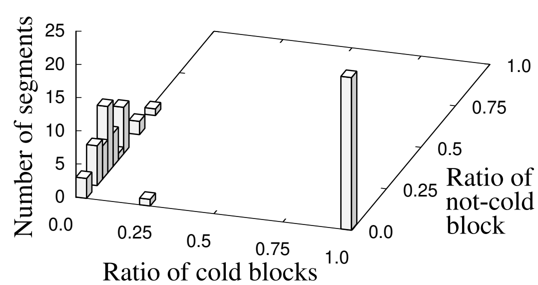

Figure 10. Distribution of cold and not-cold blocks after the 640-116 test using the separate cleaning segment.

segments for the 92-96% test; under the same load (15 segments in 2 minutes), our prototype will survive about 850 000 minutes (590 days).

4. Related Work

Logging has been widely used in certain kinds of devices; in particular, in Write-Once Read-Many (WORM) optical disk drives. WORM media are especially suitable for logging because of their append-only writing. OFC [8] is a WORM-based file system that supports a standard UNIX file system transparently. Although its data structures differ from those of our prototype, our prototype's block-address translation scheme is very similar to that of OFC. OFC is self-contained in that it stores all data structures on a WORM medium and needs no read-write medium such as a HDD. To get around the large memory requirement, it manages its own cache to provide efficient access to the structure. Our prototype, however, needs improvement with regard to its memory requirement (about 260 Kbytes for a 16-Mbyte flash memory system).

LFS [3] uses the logging concept for HDDs. Our prototype is similar to LFS in many aspects, such as segment, segment summary, and segment cleaner, but LFS does not use block-address translation. LFS incorporates the FFS index structure into the log so that data retrieval can be made in the same fashion as in the FFS. That is, each *inode* contains pointers that directly point to data blocks. Our prototype, on the other hand, keeps a log of physical block modification.

LFS gathers as many data blocks as possible to be written in order to maximize the throughput of write operations by minimizing seek operations. Since flash memory is free from seek penalty, maximizing the write size does not necessarily improve performance.

The paper on BSD-LFS [4] reports that the effect of the cleaner is significant when data blocks are updated randomly. Under these conditions, each segment tends to contain fewer invalid data blocks and the cleaner's copying overhead accounts for more than 60% of the total writes. With our prototype, this overhead accounts for about 70% on the 90%-utilized file system.

Since flash memory offers a limited number of write/erase cycles on its memory cell, our driver requires the block translation mechanism. Logical Disk (LD) [9] uses the same technique to make a disk-based file system log-structured transparently. Although the underlying storage media and goals are different, both the driver and LD function similarly. LD does, though, provides one unique abstract interface called *block lists*. The block lists enable a file system module to specify logically related blocks such as an *inode* and its indirect blocks. Such an interface might be useful for our driver by enabling it to cluster hot and cold data.

Douglis et al. [10] have examined three devices from the viewpoint of mobile computing: a HDD, a flash disk, and a flash memory. Their simulation results show that the flash memory can use 90% less energy than a disk-based file system and respond up to two orders of magnitude faster for read but up to an order of magnitude slower for write. They also found that, at 90% utilization or above, a flash memory erasure unit that is much larger than the file system block size will result in unnecessary copying for cleaning and will degrade performance.

The flash-memory-based storage system eNVy [11] tries to provide high performance in a transaction-type application area. It consists of a large amount of flash memory, a small amount of battery-backed SRAM for write buffering, a large-bandwidth parallel data path between them, and a controller for page mapping and cleaning. In addition to the hardware support, it uses a combination of two cleaning policies, *FIFO* and *locality gathering*, in order to minimize the cleaning costs for both uniform and hot-and-cold access distribution. Simulation results show that at a utilization of 80% it can handle 30 000 transactions per second while spending 30% processing time for cleaning.

Microsoft Flash File System (MFFS) [2] provides MS-DOS-compatible file system functionality with a flash memory card. It uses data regions of variable size rather than data blocks of fixed length. Files in MFFS are chained together by using address pointers located within the directory and file entries. Douglis et al. [10] observed that MFFS write throughput decreased significantly with more cumulative data and with more storage consumed.

SunDisk manufactures a flash disk card that has a small erasure unit, 576 bytes [12]. Each unit takes less time to be erased than does Intel’s 16Mbit flash memory. The size enables the card to replace a HDD directly. The driver of the card erases data blocks before writing new data into them. Although this erase operation reduces the effective write performance, the flash disk card shows stable performance under high utilization because there is no need to copy live data [10]. In a UNIX environment with FFS, simply replacing the HDD with the flash disk would result in unexpected short life because FFS meta data such as *inode*s are located at fixed blocks and are updated more often than user data blocks. The flash disk card might perform well in the UNIX environment if a proper wear-leveling mechanism were provided.

5. Conclusion

Our prototype shows that it is possible to implement a flash-memory-based file system for UNIX. The benchmark results shows that the proposed system avoids many of the problems expected to result from flash memory's overwrite incapability.

The device driver approach makes it easy to implement this prototype system by using the existing FFS module. But because the FFS is designed for use with HDD storage, this prototype needs to use a portion of the underlying flash memory to hold data structures tuned for a HDD. Furthermore, the separation of the device driver from the file system module makes the prototype system management difficult and inefficient. For example, there is no way for the driver to know whether or not a block is actually invalid until the FFS module requests a write on the block—even if the file for which the block was allocated had been removed 15 minutes before. A file system module should therefore be dedicated to flash memory.

Acknowledgments

We thank the USENIX anonymous referees for their comments, Douglas Orr for valuable suggestions and comments on the drafts, Fred Douglis for making his draft available to us, and Satyanarayanan-san and the Information Technology Center, Carnegie-Mellon University for providing us the Andrew Benchmark.

References

[1] Advanced Micro Devices, Inc., “Am29F040 Datasheet”, 1993.

[2] *Flash Memory*, Intel Corporation, 1994.

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

[4] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, “An Implementation of a Log-Structured File System for UNIX”, Proc. ’93 Winter USENIX, 1993.

[5] C. Ruemmler and J. Wilkes, “UNIX disk access patterns”, Proc. ’93 Winter USENIX, 1993.

[6] J. H. Howard, et al., “Scale and Performance in a Distributed File System”, ACM Transactions on Computer Systems, Vol. 6, No. 1, 1988.

[7] M. K. McKusick, M. J. Karels, and K. Bostic, “A Pageable Memory Based Filesystem”, Proc. ’90 Summer USENIX, 1990.

[8] T. Laskodi, B. Eifrig, and J. Gait, “A UNIX File System for a Write-Once Optical Disk”, Proc. ’88 Summer USENIX, 1988.

[9] 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.

[10] F. Douglis, R. Cáceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, “Storage Alternatives for Mobile Computers”, Proc. 1st Symposium on Operating Systems Design and Implementation, 1994.

[11] M. Wu and W. Zwaenepoel, “eNVy: A Non-Volatile, Main Memory Storage System”, Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[12] “Operating system now has flash EEPROM management software for external storage devices” (in Japanese), Nikkei Electronics, No. 605, 1994.

Author Information

Atsuo Kawaguchi is a research scientist at the Advanced Research Laboratory, Hitachi, Ltd. His interests include file systems, memory management system, and microprocessor design. He received B.S., M.S., and Ph.D. degrees from Osaka University. He can be reached at atsuo@harl.hitachi.co.jp.

Shingo Nishioka is a research scientist at the Advanced Research Laboratory, Hitachi, Ltd. His interests include programming languages and operating systems. He received B.S., M.S., and Ph.D. degrees from Osaka University. He can be reached at nis@harl.hitachi.co.jp.

Hiroshi Motoda has been with Hitachi since 1967 and is currently a senior chief research scientist at the Advanced Research Laboratory and heads the AI group. His current research includes machine learning, knowledge acquisition, visual reasoning, information filtering, intelligent user interfaces, and AI-oriented computer architectures. He received his B.S., M.S., and Ph.D. degrees from the University of Tokyo. He was on the board of trustees of the Japan Society of Software Science and Technology and of the Japanese Society for Artificial Intelligence, and he was on the editorial board of Knowledge Acquisition and IEEE Expert. He can be reached at motoda@harl.hitachi.co.jp.

All the authors can be reached at

Advanced Research Laboratory, Hitachi, Ltd.
Hatoyama, Saitama, 350-03 Japan.