Brief Contributions

A Hybrid Flash File System Based on NOR and NAND Flash Memories for Embedded Devices

Chul Lee, *Student Member*, *IEEE*,
Sung Hoon Baek, *Student Member*, *IEEE*, and
Kyu Ho Park, *Member*, *IEEE*

Abstract—This paper presents a hybrid flash file system (HFFS) based on both NOR flash and NAND flash memory. In a conventional NAND flash-based flash file system, there is a trade-off between life span and durability in the frequent writing of small amounts of data. Because NAND flash supports only a page-level I/O, at least one page is wasted in the synchronous writing of small amounts of data. The wasting of pages reduces the utilization and life span of the NAND flash. To alleviate the utilization problem, some NAND flash-based flash file systems write small amounts of data asynchronously with RAM buffers, though buffering in RAM decreases the durability of the system. Our HFFS eliminates the trade-off between life span and durability. It synchronously stores data as a log in the NOR flash, whenever we append small amounts of data to a file. The merged logs are then flushed to the NAND flash in a page-aligned fashion. The implementation of our HFFS is based on our previous NAND flash-based file system, called CFFS [1]. The experimental results reveal that our HFFS provides a longer life span than a conventional NAND flash-based synchronous flash file system with a similar level of durability.

Index Terms—Storage management, file system management, NOR flash, NAND flash, embedded device.

1 Introduction

THERE are two types of flash memory: NOR flash and NAND flash. Both types of flash memory are widely used as a storage medium of embedded systems such as sensor devices, mobile devices, and event data recorders because they provide affordable capacity, shock resistance, and low power consumption.

The two types are distinctive in terms of density, performance, and operating characteristic [2], [3]. The density of the NAND flash is much greater than that of the NOR flash. In terms of performance, both types are comparable except with regard to the erase time. The erase time of the NOR flash (around 0.7 second) is 350 times longer than the one of the NAND flash (2 ms). However, the NOR flash supports byte-addressable access, but the NAND flash does not. The NOR flash is usually used in a system that needs to boot out of flash, execute code from flash, and store only small amounts of data because it supports byte-addressable operations and a fast read speed. Furthermore, because the NAND flash provides a higher capacity and a faster write speed than the NOR flash, the NAND flash is widely used for data storage applications.

However, the NAND flash can be read or programmed only in the unit of a page (512 bytes or 2 Kbytes). Once a page is programmed, it cannot be overwritten before being erased. Hence, at least one page is consumed for writing even small amounts of data.

In embedded systems, the major file operations are the updating or appending of small amounts of data, yet most of the capacity is occupied by large files. For instance, sensor devices or an event data recorder [4], [5], [6] periodically collect a few bytes of sensed data, but store large amounts of historic data over long periods. In a mobile phone, small records such as text messages and an address book are updated sporadically, while comparatively large images and media files occupy most of the storage area. Other embedded systems, such as manufacturing systems, frequently update configuration files and parameters and maintain a large log of the system's status.

A typical design of the storage system in a sensor or embedded device is shown in Fig. 1. As shown in Fig. 1a, the first generation of the devices was equipped only with the NOR flash. NOR flash provides one storage region for the codes and another for the data. However, as the collected data accumulates, a greater storage capacity is required. The NAND flash is therefore applied to the devices. Recent devices, as shown in Fig. 1b, combine the NOR flash with the NAND flash. Because of its unique characteristics, the NOR flash is generally used for storing boot codes because it supports execution in-place. The NAND flash, on the other hand, is used for storing the collected event data. However, because of the page-based I/O nature of a NAND flash, applications that frequently update records that are much smaller than the size of a page of a NAND flash greatly reduce the utilization of the medium. Moreover, low utilization deteriorates the life span of flash memory.

To alleviate the low-utilization problem in a NAND flash-based file system, small amounts of data are written asynchronously after being merged in a RAM buffer. The asynchronous writing prolongs the life span but degrades the durability because it increases the possibility of data being lost in a crash. Because embedded systems are likely to suffer from uncertain power outages, data durability and reliability are critical features of those systems.

As shown in Fig. 1c, nonvolatile RAM, such as a battery backed-up SRAM (BBSRAM) [7], can then be used to merge the small amount of data while ensuring durability. However, it is a pseudo-nonvolatile RAM that needs to be continuously recharged in order to ensure data retention. In addition, because of the presence of the battery, the BBSRAM has numerous weaknesses in terms of cost, humidity, shock, vibration, space, and maintenance.

We propose a hybrid flash file system (HFFS) which combines both types of flash memory in the file system layer for reliable and durable embedded systems like Fig. 1d. The HFFS synchronously stores a small amount of data as a log in the NOR flash because the NOR flash is byte addressable. When the NOR flash logs enough data to fill a page of NAND flash, the data are merged and copied in the NAND flash. The HFFS therefore does not generate excessive garbage in the NAND flash in the synchronous appending of a small amount of data. Our HFFS also provides a single combined partition for applications without distinguishing between the NOR flash and the NAND flash. Hence, the HFFS simplifies applications in the use of both the NOR flash and the NAND flash.

Our HFFS eliminates the trade-off between life span and durability; it provides the same level of durability as a conventional synchronous NAND flash-based flash file system but also supports a prolonged life span.

• *The authors are with the Department of Electrical Engineering, Korea Advanced Institute of Science and Technology, 373-1 Gusung-dong Yousung-gu, Daejon, Korea 305-701.*
E-mail: {chullee, shbaek}@core.kaist.ac.kr, kpark@ee.kaist.ac.kr.

Manuscript received 21 May 2007; revised 2 Nov. 2007; accepted 15 Nov. 2007; published online 8 Jan. 2008.
Recommended for acceptance by J. Antonio.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-05-0175.
Digital Object Identifier no. 10.1109/TC.2008.14.

0018-9340/08/$25.00 © 2008 IEEE Published by the IEEE Computer Society

INTEL-1014
9,134,190

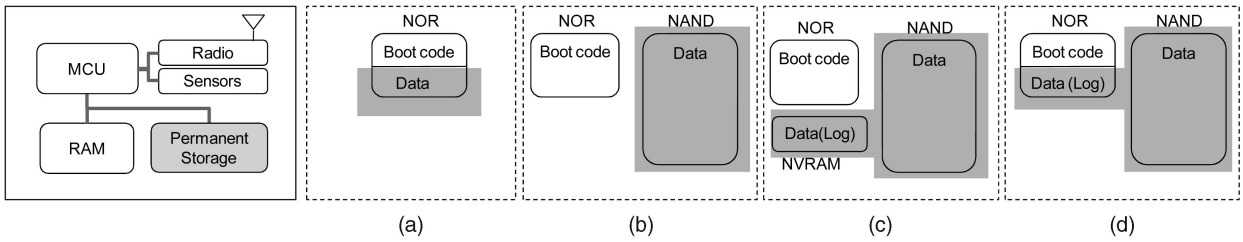

Fig. 1. Storage alternatives for embedded devices: (a) Only NOR flash, (b) NOR flash and NAND flash for storing codes and data, (c) NAND flash with a nonvolatile RAM, and (d) proposed hybrid architecture in which logs of small updates are stored in the NOR flash and others are stored in the NAND flash.

2 Background and Related Works

In this section, we present a brief overview of flash memory constraints and previous works on flash-based storage systems.

2.1 Flash Memory Constraints

Flash memory has three physical constraints. The first constraint is a constraint in which an in-place update is not allowed because any written areas should be erased before they can be reprogrammed. Out-of-place updates are responsible for most of the constraints in the design of flash filing systems. The second constraint is a size problem in which the size of the erase unit is much larger than that of a program unit. The erase unit is called an erase block. In a recent flash chip, the size of an erase block is 128 Kbytes, whereas the write unit is a word or a page of 2 Kbytes. Thus, live data and obsolete data may exist in an erase block. A cleaning operation like garbage collection is then required to move the live data into a free erase block before the erasure. The third constraint is the limited life span of an erase block due to the number of erase counts. Usually 100,000 erase cycles are guaranteed for each erase block. The erase cycles should be even throughout all of the erase blocks in order to level the wear of the erase blocks.

2.2 Flash-Based Storage System

Many works have focused on efficient flash-based storage systems and they can be classified into two main strategies. The first strategy involves the creation of a virtual block device layer, called a flash translation layer [8], between a legacy file system and flash chips. The flash translation layer hides the flash memory constraints from the legacy file system. The other strategy is to implement a flash-aware file system which is designed specifically for direct use on flash chips without translation layers. The flash-aware file system, which differs from legacy file systems, should be aware of the flash properties of out-of-place updates.

Many small embedded devices in a ubiquitous sensor network are equipped with flash memories and flash-aware file systems. The ELF [9], MicroHash [10], and TINX [11] systems are designed to store sensed data in the NAND flash of sensor devices. The most important design constraints are a small memory requirement and low power consumption.

In addition, the JFFS2 [12] and YAFFS [13] systems were designed for general embedded systems equipped with several megabytes of NAND flash. These designs, which are based on a traditional log structured file system [14], can perform out-of-place updates and cleaning. However, the JFFS2 and YAFFS systems were designed for small flash memory of less than several megabytes. For these systems, the entire flash medium should be scanned at the mounting time. Moreover, because the scanning time is proportional to the size of the flash memory, the time required for mounting a large flash memory would be intolerable. Another consideration is the fact that every file abstraction and data structures for resource management should always be loaded into the RAM. In terms of mounting time and memory requirements, the previous JFFS2 and YAFFS systems are not scalable. In JFFS2, most of the NAND flash area should be scanned to load file system-specific data structures in the RAM. However, YAFFS scans the entire spare region for loading file system structures and part of the data regions that contain metadata. To alleviate the scalability problem for large flash memories, researchers on CFFS [1] and JFFS3 [15] have proposed a reduction in the scanning time and memory requirements.

However, the previous NAND flash-based flash file systems failed to consider the durability and life span of frequent small updates and there was a trade-off between durability and the life span. For a durable system, the file system should be mounted as a synchronous mode, resulting in the wasting of a single page of the NAND flash for the writing of even a small amount of data. Asynchronous writing, on the other hand, sacrifices data durability.

3 NOR and NAND Flash-Based Hybrid Flash File System

In this section, we describe the architecture, data structures, and operations of our proposed HFFS. We also present an analysis of the life span and the threshold size that determines whether data is logged to the NOR flash or stored in NAND flash. Finally, we discuss performance issues with respect to the shortcomings of the NOR flash.

3.1 Architecture

Fig. 2 shows how the HFFS uses both types of flash memory as a storage medium. First, NAND flash consists of multiple erase blocks, which are the unit of the erase operations. Each erase block contains a fixed number of pages, which are the unit of I/O operation. A page is a basic allocation unit and each page may contain file data or metadata. Second, the NOR flash also consists of multiple erase blocks and each erase block has multiple log

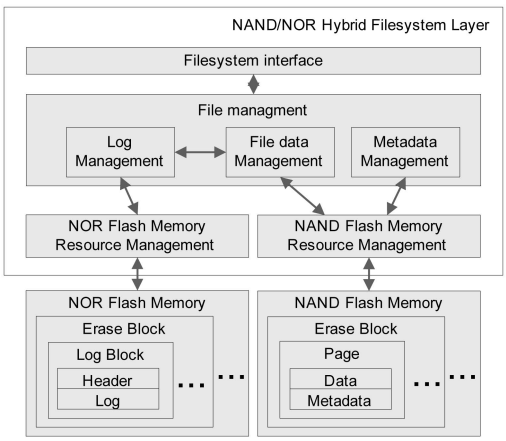

Fig. 2. The architecture of a hybrid flash file system (HFFS).

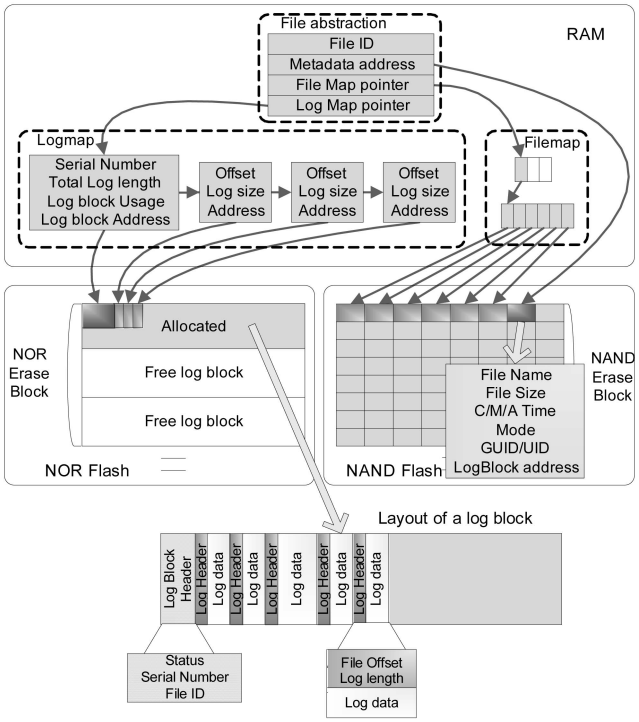

Fig. 3. Data structures in the RAM, NOR flash, and NAND flash.

blocks. A log block, which is reserved for a file and contains multiple variable-sized logs that correspond to the file, is a basic allocation unit for the NOR flash. The size of the log block is defined when the file system is created.

There are two resource management modules for both types of flash memory. The resource management covers the management of erase blocks, garbage collection, and the allocation of free pages or log blocks.

Other than these two resource management modules for both types of flash memory, there is a file management module. For each file, there are three corresponding types of data: metadata, file data, and logs. The file management module should maintain the metadata, file data, and logs of each file.

The file metadata is stored in the NAND flash, but the file data can be written in either the NAND flash or the NOR flash. When the file data is small enough, it is transformed to a log and the log is written in the NOR flash. Otherwise, the file data is written in the NAND flash.

3.2 Data Structures

The data structures of the HFFS are based on a previous NAND flash-based file system, the CFFS [1]. The CFFS inherited the data structures of NAND flash, though we designed the log-related data structures of the NOR flash for our proposed system. Fig. 3 shows the data structures of the RAM, NOR flash, and NAND flash; the arrow lines highlight the relations between each of the data structures.

Each file abstraction has pointers to a file map and a log map. The file map maintains addresses to physical pages of data in the NAND flash. The log map keeps track of logs that are written in the NOR flash. Furthermore, each file has corresponding metadata in the NAND flash and the page address of the metadata is kept in the file abstraction. The metadata page contains the name, size, and other attributes of the file. The file abstraction and the file map are constructed in the RAM at the mount time, as with the CFFS. However, the log map is constructed in the RAM upon the first access by scanning the corresponding log block.

```
typedef struct {
    __u8 status;
    __u8 serial;
    __u32 objectId;
} hffs_LogBlockHeader;

typedef struct {
    __u32 offset;
    __u16 length;
} hffs_LogHeader;
```

(a)

```
typedef struct {
    __u8 serial;
    __u32 totLogLen;
    __u32 bytesInUse;
    __u32 objectId;
    struct list_head logList;
} hffs_LogBlockDesc;

typedef struct {
    __u32 offsetInFile;
    __u16 length;
    __u32 offsetInBlock;
    struct list_head list;
} hffs_LogDesc;
```

(b)

Fig. 4. Main data structures used in our HFFS: (a) Data structures of a log block header and a log header and (b) data structures for a log map.

The layout of a log block is shown at the bottom of Fig. 3. The log block header leads and the multiple logs follow. The log block header includes a serial number, a file ID, and a status field. As a means of resolving ambiguity between an old log block and a new log block, the serial number is incremented whenever a log block is allocated. The status field is needed to identify the status of a log block. A log consists of a log header and log data. The log header consists of the size of the log and the file offset that describes the position of the log in the file. Fig. 4a shows the data structures of a log header and a log block header.

The log map describes the log block and maintains a list of logs with the two data structures shown in Fig. 4b, namely, a log block descriptor and a log descriptor. The log block descriptor contains the total log length and fullness of the log block. In addition, each log descriptor has the log's size, file offset, and location inside the log block.

When a small amount of data is first appended to a file, an empty log block is allocated to the file. At that time, a log map is created in the RAM with a log list that includes the new log. A subsequent small amount of data is subsequently appended to the tail of the log block as a form of log. Furthermore, the corresponding log descriptor is appended to the tail of the log list of the log map. When the log block is filled with enough logs, every log is flushed to the NAND flash and the corresponding log descriptors are freed from the RAM.

3.3 File Operations

File operations normally follow the same process as in the previous implementation of the CFFS so that access is directed just to the NAND flash. However, when a file is associated with logs, the operations become slightly complex. The file operations are described in the following.

3.3.1 Reading a File

Fig. 5 gives a description of the flow of a read operation. When a file is first accessed, the file's metadata is read as a means of checking whether the file has logs in the NOR flash. If the log block address in the metadata is invalid, the file does not have logs in the NOR flash and the read operation is subsequently directed to the NAND flash. On the other hand, if the file has logs in the NOR flash, the log block is scanned and the corresponding log map is loaded in the RAM before the data or attributes of the file are read. Because a log header contains the length of the following log data, we can locate the next log header. Furthermore, by reading all of the log headers, we can construct a log map that is a doubly linked list of log descriptors. When the log map is completely loaded in the RAM, the requested read operation is serviced by the reading of logs that are described in the linked list of logs.

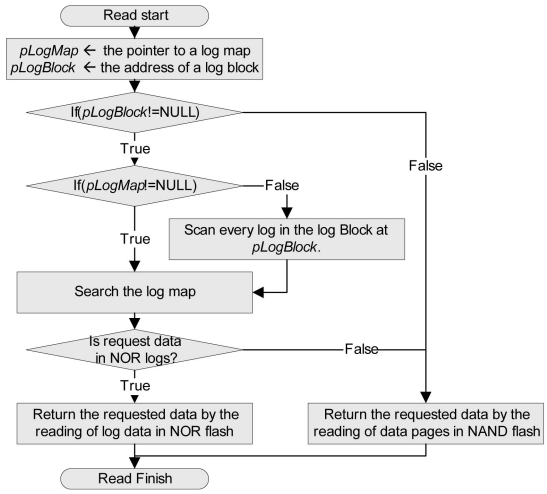

Fig. 5. The flow of a read operation.

3.3.2 Appending a Small Amount of Data to a File

Fig. 6 shows the flow of a write operation. When an application appends a small amount of data to a file, the size of the data is checked first. Only data that is smaller than a predetermined threshold is stored as a log in the NOR flash; larger amounts of data are stored in the NAND flash. The threshold of the data size was determined by using life span analysis, as explained in Section 3.3.5.

To write file data as a log, we need to check if the log block address in the metadata is valid. If the log block address is invalid, the file does not have logs in the NOR flash; hence, a new empty log block is allocated to the file and a new log map is constructed in the RAM.

If the log block address is valid, the file has logs in the NOR flash; hence, the log block should be scanned and the corresponding log map should be loaded in the RAM before the data is appended.

Finally, the new data is appended at the tail of the log block, a new descriptor of the log is appended to the log map in the RAM, and the file metadata is updated with an address of the newly allocated log block.

However, while appending logs, the log block grows and is filled with logs. When the log block is filled with enough logs, all of the logs in the log block are transferred to data pages of the NAND flash. We refer to this job as *flushing*. The flushing job includes the loading of file data from the log block to RAM, the allocation of free data pages in the NAND flash, the writing of loaded file data to the NAND flash, and the making of an obsolete mark for the flushed log block. The log block eventually becomes obsolete and can be erased later.

3.3.3 Cleaning Obsolete Erase Blocks

An obsolete erase block of the NOR flash should be erased at the proper time before it is needed. Given that a block erase time of the NOR flash is around 0.7 second, we could not perform the block erase operation synchronously to support real-time capability. Instead, we run a cleaning thread in the background. Whenever the cleaning thread finds an obsolete erase block, it performs an erase operation on the obsolete block. However, the read or write requests are capable of preempting the cleaning thread. When an erase operation is in progress, the erase operation can be suspended to perform a read or write operation and the erase operation will be resumed later.

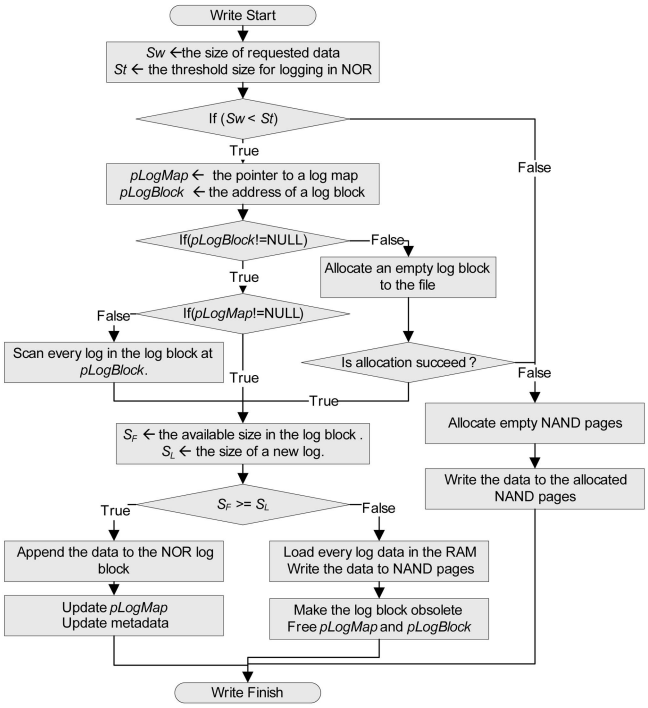

Fig. 6. The flow of a write operation.

3.3.4 Ensuring Consistency and Recovering from a Crash

Our HFFS synchronously writes the log in the NOR and atomically flushes the merged logs in the NAND; thus, there is no consistency problem between logs in the NOR and data in the NAND. Even if two different threads try to access a single file, the writing operation of one thread is completed before the other thread accesses it. However, if a file system crashes before completing the writing of a log in the NOR, the current design of the HFFS may cause a data consistency problem, where the log header is written well, but the log data contains some dummy data. In order to remedy the above consistency problem, a *commit* mark can be written at the end of the log data.

When a file system is remounted after a crash, a recovery operation is needed to return it to a consistent state. The status flag of one byte in the log block header is used for recovery, which indicates the status of the corresponding logblock: *free (0xFF)*, *allocated (0xFC)*, *flushing (0xF0)*, and *flushed (0xC0)*. This multiple programming is possible in the NOR flash because the programming operation can change each bit from a *1* to a *0*. The status flag will be changed from *0xFF* to *0xC0* sequentially. When a file system is remounted, the NOR flash is scanned and each status flag of the logblocks is read. When the status indicates *free* or *flushed*, then the logblock was completely flushed or erased; thus, there is nothing to do. However, if the status indicates *allocated* or *flushing*, then the file system has crashed during logging in the logblock or flushing the logblock. In these cases, the log data will be loaded and merged in the RAM and the merged log data will be flushed in the NAND flash. Even though the logblock is partially flushed, the previously flushed pages in the NAND will be ignored. In this manner, our HFFS provides atomicity in flushing the merged logs in the NAND flash.

3.4 Analysis of the Life Span of NOR and NAND Flash

To determine the threshold size of a log, we performed an analysis of the life spans of the NOR flash and NAND flash. Because one of our goals is to prolong life span, we determined the expected life spans of both the conventional NAND flash-based file system and our proposed HFFS.

Let L_h be the life span of our HFFS, which is the time until the NOR or the NAND flash totally wear. The L_h is equal to the smallest life span between both types of flash memory, $min(L_h^{nor}, L_h^{nand})$, where L_h^{nor} and L_h^{nand} are the life span for the NOR flash and NAND flash, respectively.

The parameters S_{nor}, S_{nand}, and S_{page} present the size of the NOR flash, the size of the NAND flash, and a page of the NAND flash. Let S_{ld} be the size of a log data for each update. The parameter S_{lh} is the size of a log header that includes a log length and an offset and E_{lim} is the erase cycle limit for an erase block.

Let us assume that there are only periodic small updates in the file system with a frequency of f. In addition, both flashes are assumed to have perfect wear leveling. If R_h^{nor} is the average rate of updates in the NOR flash (bytes per second), the life span of the NOR flash will be $(S_{nor} \cdot E_{lim})/R_h^{nor}$. Since $(S_{ld} + S_{lh})$ bytes of data are synchronously written with the frequency f, the rate R_h^{nor} is equal to $(S_{ld} + S_{lh}) \cdot f$ and the life span of the NOR flash can be expressed as follows:

$$L_h^{nor} = \frac{S_{nor} \cdot E_{lim}}{R_h^{nor}} = \frac{S_{nor} \cdot E_{lim}}{(S_{ld} + S_{lh}) \cdot f}. \quad (1)$$

Similarly, R_h^{nand}, the average rate of updates in the NAND flash, will be $S_{ld} \cdot f$ since the merged log data is written asynchronously. The life spans of the NAND flash can then be expressed as follows:

$$L_h^{nand} = \frac{S_{nand} \cdot E_{lim}}{R_h^{nand}} = \frac{S_{nand} \cdot E_{lim}}{S_{ld} \cdot f}. \quad (2)$$

If S_{nand} is much larger than S_{nor}, L_h is close to L_h^{nor}.

If we let L_c be the life span of the conventional NAND flash-based file system, L_c is expressed as follows:

$$L_c = \frac{S_{nand} \cdot E_{lim}}{S_{page} \cdot f} \quad (3)$$

because at least one page is consumed even for small updates.

For the HFFS to have a longer life span than the conventional file system, L_h should be larger than L_c. That is,

$$L_h \approx L_h^{nor} = \frac{S_{nor} \cdot E_{lim}}{S_{ld} + S_{lh} \cdot f} > \frac{S_{nand} \cdot E_{lim}}{S_{page} \cdot f} = L_c. \quad (4)$$

The size of a log data, S_{ld}, then becomes

$$S_{ld} < \frac{S_{nor}}{S_{nand}} \cdot S_{page} - S_{lh}. \quad (5)$$

Therefore, to prevent the NOR flash of the HFFS from being worn out sooner than the NAND flash of a conventional flash file system, only data that is sufficiently small to meet (5) should be stored as logs. For instance, when we are given a NAND flash of 128 Mbytes (with 2 Kbytes pages) and a NOR flash of 4 Mbytes, the amount of data should be less than 58 bytes because each log header includes 6 bytes. Hence, only data that totals less than 58 bytes can be stored in NOR flash. Any other data that is larger than 58 bytes is stored in the NAND flash, as with a conventional NAND flash-based file system.

3.5 Performance Issues

The NOR flash has faster access but a much slower write and erase time than the NAND flash. We discuss the performance shortcomings of the NOR flash in terms of the erase latency, the write latency, the flushing time, and the wear leveling.

3.5.1 Erase Latency

The NOR flash has a much logner erase time than the NAND flash. Erasing one block of 64 Kbytes takes around 0.7 second in a typical NOR flash [3] but 2 milliseconds in a recent NAND flash [2]. The very long erasing time greatly reduces the write throughput and the real-time performance of the NOR flash.

However, we can hide the long erase delay by running a cleaning thread in the background. Recent flash chips support two advanced features to provide preemptiveness of an erase operation, namely, the *erase suspend/resume* feature and the *simultaneous read/write* feature [16].

Use of the *erase suspend/resume* enables an erase operation to be interrupted and paused so that data can be accessed from a block that is not being erased. When a read or a write request interrupts an erase operation, the erase operation is suspended until the request is serviced completely. However, although the *erase suspend/resume* provides the preemptiveness of an erase operation, it increases the total erase time. To erase a block, a flash memory issues a number of erase pulses. When an erase operation is suspended, any incomplete erase pulses should be restarted. The *erase suspend/resume* feature slightly reduces the erase efficiency and increases the erase time.

Flash chips with a *simultaneous read/write* feature enable a read operation to occur while an erase or program operation is being executed. As a result, the erase time is not prolonged. However, this feature does not enable simultaneous erasing and programming; hence, a program operation is incapable of preempting an erase operation.

3.5.2 Write Latency

Generally speaking, the write latency of NOR flash is longer than that of NAND flash, but it depends on the size of the written data. Because NOR flash is byte addressable, the write latency is linearly proportional to the size of the written data. However, the fixed time for programming one page elapses whenever a small amount of data is written in the NAND flash. For instance, the time taken to write one word (2 bytes) is almost the same as the time taken to write one page (2 Kbytes) in the NAND flash. The write latencies for writing data of less than 32 bytes are comparable in both types of flash memory.

3.5.3 Flushing

The flushing process blocks other read or write requests until it is completed because the flushing should be performed atomically for consistency. The delay depends on the size of a log block. As a log block becomes larger, the flushing is deferred further and the flushing delay is prolonged.

Let T_f be the time taken for the flushing. Because flushing consists of reading a log block from the NOR flash and writing merged data to the NAND flash,

$$T_f = tr_{lb}^{nor} + tw_p^{nand} \cdot N_p, \quad (6)$$

where tr_{lb}^{nor} is the time taken to read a log block from the NOR flash, tw_p^{nand} is the time taken to write a page in the NAND flash, and N_p is the number of pages to be written in the NAND flash.

Since $N_p = \lceil \frac{S_{ld}}{S_{ld}+S_{lh}} \cdot S_{lb}/S_{page} \rceil$,

$$T_f = tr_{byte}^{nor} \cdot S_{lb} + tw_p^{nand} \cdot \left\lceil \frac{S_{ld}}{S_{ld} + S_{lh}} \cdot \frac{S_{lb}}{S_{page}} \right\rceil. \quad (7)$$

As described in (7), the flushing delay is mainly proportional to the size of a log block. For instance, it takes 13.7 milliseconds to flush a log block of 64 Kbytes, which contains logs of 16 bytes and log headers of 6 bytes. Therefore, the size of a log block should be carefully chosen in order to meet a given real-time system requirement.

3.5.4 Wear-Leveling Effect

In our HFFS, every small item of data is sequentially logged in the NOR flash just until it is flushed to the NAND flash. Thus, a complex wear-leveling scheme is unnecessary for the NOR flash. Instead, we allocated an empty erase block in a sequential order, thereby enabling the wear of the erase blocks to be leveled naturally. Furthermore, the HFFS provides a better garbage collection performance of the NAND flash than a conventional

TABLE 1
Experimental Setup for the Emulation of NOR and NAND Flash

Flash	NOR	NAND
Size	4 MB	128 MB
Page size	(n/a)	2 KB
Log block size	64 KB	(n/a)
Read	90 ns @ 2 B	125 μs @ 2 KB
Write	11.5 μs @ 2 B	400 μs @ 2 KB
Erase	0.7 s @ 64 KB	2 ms @ 128 KB
Erase Cycle Limit	50 cycles	50 cycles

flash file system, which generates excessive garbage while synchronously appending frequent small amount of data.

4 Evaluation

We used Linux 2.6 to implement our CFFS-based HFFS. Even though our CFFS outperforms the conventional NAND flash-based file system such as the YAFFS in the aspect of scalability, the schemes of writing small amounts of data are similar to each other; both of them showed similar results in the following experiments. Thus, we compared the results of HFFS with those of the YAFFS, which is more popular.

Both types of flash memory are emulated in the RAM regions. We used *nandsim* and *mtdram* [17] for the NAND and the NOR emulation, respectively. In addition, as shown in Table 1, we modeled delay factors in our emulated flash chips based on the datasheets of real chips [2], [3]. The emulated memories sleep during the requested operations, enabling us to measure the I/O performance. Furthermore, our emulated memories are modified to be capable of counting the number of erase operations on each block. Hence, we can measure the wear of the erase blocks.

We synthesized two workloads of an automotive black box [6] and a mobile phone. First, the automotive black box periodically records event data such as velocity, engine speed, and wheel positions, which is very useful for future reference after a collision. Our synthesized workload collects event data of 16 bytes every 10 milliseconds and synchronously appends the data to a file. Second, the daily usages of a mobile phone are simulated. It appends 16 bytes of entries to three files for the last dialed numbers, received calls, and missed calls. In addition, two message files with variable-length messages, ranging from 16 to 96 bytes, are stored for incoming and outgoing messages. During the simulation of 1,000 days, the phone receives and sends 20 messages per day, receives 20 and dials 20 calls, misses five calls, and cumulates 64 Mbytes of media files.

The life span of our HFFS is measured and compared with that of the YAFFS. We also introduce a flash utilization to see how much excessive garbage is reduced. Next, we discuss durability issues. In addition, we measured the write performance to see if the HFFS overcomes the performance shortcomings of NOR flash and we compared the wear leveling performance between our HFFS and the YAFFS.

4.1 Life Span

To measure the life span of our HFFS and the YAFFS, we configured the synchronous mode of the YAFFS to synchronously write data to the NAND flash, thereby ensuring that both systems had the same level of data durability.

The erase cycle limit is reduced to 50 cycles for both emulated flash memories because it would take years to complete the experiment with 100,000 erase cycles. However, 50 cycles seems to be adequate for comparing relative results.

In the automotive black box workload, our HFFS lives four times longer than the YAFFS, as shown in Fig. 7. The life span is around 9 hours for YAFFS with only NAND flash but around 36 hours in the HFFS. When we set the erase cycle limit at 100,000 cycles, the life span is only two years (18,000 hours) for the YAFFS, but 8 years in the HFFS.

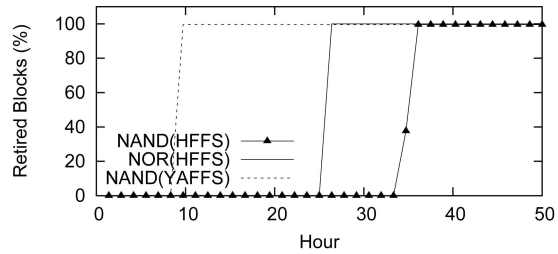

Fig. 7. Percentage of retired erase blocks when appending a log of 16 bytes every 10 ms.

4.2 Excessive Garbage Reduction

In order to see how much excessive garbage is generated, we introduce a new metric called a flash utilization U_f, which is $\frac{The\ size\ of\ total\ written\ data}{The\ size\ of\ total\ programmed\ area}$. Thus, the greater U_f means that less excessive garbage is generated. In Table 2, the flash utilizations are listed for the YAFFS under synchronous mode, the YAFFS under asynchronous mode, and our HFFS, when we simulated a mobile phone workload. The size of total written bytes is 70.25 Mbytes in three systems, but the sizes of the total programmed areas are largely different. The YAFFS under synchronous mode programmed over 330 Mbytes, including the copying of the valid pages for garbage collection; however, the YAFFS under asynchronous mode programmed only around 72 Mbytes because small updates are merged in the RAM. Our HFFS programmed around 72 Mbytes in the NAND flash and around 4 Mbytes in the NOR flash. In our HFFS, small updates are merged in the NOR flash and they are flushed in the NAND flash again; therefore, the writing of a small amount of data is duplicated in the NOR and NAND. The U_f of our HFFS is slightly less but is close to that of the YAFFS under asynchronous mode.

4.3 Durability

We compared how long the data resides in the RAM before being written in the NAND flash. First, we configured the asynchronous mode of the YAFFS to buffer a small amount of data in the RAM. We then made a buffer of 2 Kbytes, which is the size of a NAND page. When the buffered data reached 2 Kbytes, the data were subsequently flushed to the NAND flash. Note that the buffer delay is related to the data writing rate and the size of each datum. For example, any written data can reside in a volatile RAM for up to two seconds in an automotive black box application that writes 16 bytes of sensed data every 10 milliseconds. Thus, data acquired in two seconds is likely to be lost if the system suffers an unexpected power outage. The loss of data for that two seconds will be critical in systems that require high reliability, such as an automotive black box or an accounting system. Our HFFS, on the other hand, synchronously writes any small amount data in the nonvolatile flash memory and the data can be recovered in the event of an unexpected power outage.

4.4 Write Latency

Fig. 8 shows the write latencies as the system continues to append 16 bytes to a file. We ran the YAFFS in a synchronous mode. In the

TABLE 2
Flash Utilization (PB_{nand} and PB_{nor} Are the Total Programmed Bytes in the NAND and NOR Flash)

	YAFFS_sync	YAFFS_async	HFFS
Written bytes	70.25 MB		
PB_{nand}	330.83 MB	72.44 MB	72.25 MB
PB_{nor}	-	-	4.15 MB
Utilization(U_f)	21.2 %	97.0 %	92.0 %

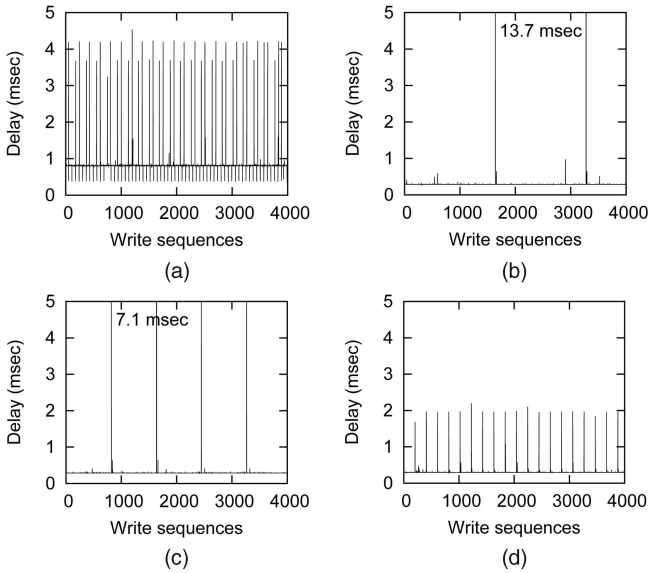

Fig. 8. Write delays of writing 16 bytes per each sample: (a) YAFFS, (b) HFFS with the 64-Kbytes NOR log block, (c) HFFS with the 32-Kbytes NOR log block, and (d) HFFS with the 8-Kbytes NOR log block.

YAFFS, the garbage collection is triggered to perform frequently because small appendages frequently generate garbage. Each garbage collection process takes 4 milliseconds.

In most instances, the write latencies of the HFFS are smaller than those of the YAFFS. The HFFS merges a small amount of data and stores the merged data in the NAND flash in a page-aligned manner; thus, small appendages do not generate any garbage. Hence, we could not observe any delay due to garbage collection. However, as shown in Fig. 8b, we observed the longest delay when the log block was as large as 64 Kbytes. This delay is caused by a prolonged flushing job whereby a considerable amount of appended data is copied from the NOR flash to the NAND flash. To minimize the delay, we should limit the size of a log block, which is the unit of a flushing job. When we set the log block size to 32 Kbytes, as shown in Fig. 8c, the worst-case delay was reduced to around 7 milliseconds.

4.5 Wear Leveling Effect

To see the wear leveling performance, we increased the erase cycle limit to 100,000 cycles. We then expanded a file by appending 16 bytes for each sample until the size of the file reached 64 Mbytes, which is half of the NAND flash size. Because of the problem of excessive garbage, the YAFFS performs many garbage collection operations. As shown in Fig. 9, the erase cycles in the NAND flash range from 62 to 176.

The HFFS does not generate garbage because it merges a small amount of data in the NOR flash. The erase count of the NAND flash is at most one cycle over the whole erase blocks, whereas the erase count of the NOR flash is less than 50 cycles. Moreover, the wear level is distributed evenly over all of the erase blocks in both the NOR flash and the NAND flash.

5 Conclusion

We designed and implemented HFFS based on NOR flash and NAND flash. In a conventional NAND flash-based flash file system, there is a trade-off between the life span and durability. Because NAND flash supports only a page-level I/O, a single page of 2 Kbytes is required in order to synchronously write a small amount of data.

Our HFFS prolongs the life span and enhances the durability of written data more effectively than conventional NAND flash-based file systems. When we append a small amount of data to a file, the data are synchronously stored as a log in the NOR flash. The merged logs are then flushed to the NAND flash in a page-aligned fashion.

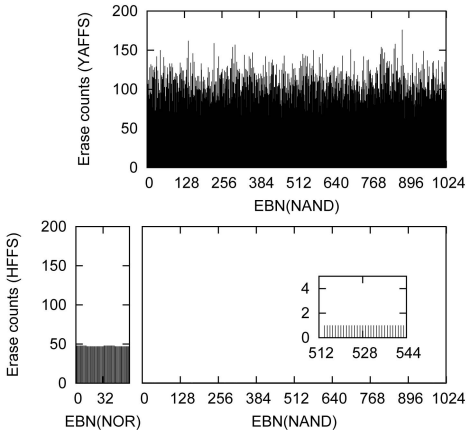

Fig. 9. Erase counts over erase blocks of NOR and NAND flash (EBN is the erase block number).

By doing this, we avoid data loss in the event of an unexpected power outage. Furthermore, by increasing the utilization of NAND flash, we can prolong the life span than a conventional NAND flash-based file system. Finally, the HFFS provides a single combined partition to facilitate its usage in applications.

References

[1] S.H. Lim and K.H. Park, "An Efficient NAND Flash File System for Flash Memory Storage," *IEEE Trans. Computers*, vol. 55, no. 7, pp. 906-912, July 2006.

[2] SAMSUNG Electronics, "K9F1G08, 128M × 8 Bit NAND Flash Memory," http://www.samsung.com/Products/Semiconductor/NANDFlash, 2008.

[3] SAMSUNG Electronics, "K8S3215, 32M Bit (2M × 16) NOR Flash Memory," http://www.samsung.com/Products/Semiconductor/NORFlash, 2008.

[4] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An Analysis of a Large Scale Habitat Monitoring Application," *Proc. Second Int'l Conf. Embedded Networked Sensor Systems*, pp. 214-226, 2004.

[5] R. Jafari, A. Encarnacao, A. Zahoory, F. Dabiri, H. Noshadi, and M. Sarrafzadeh, "Wireless Sensor Networks for Health Monitoring," *Proc. Second Ann. Int'l Conf. Mobile and Ubiquitous Systems: Networking and Services*, 2005.

[6] U.S. Dept. Transportation, Nat'l Highway Traffic Safety Administration, "Event Data Recorder(EDR) Applications for Highway and Traffic Safety," http://www-nrd.nhtsa.dot.gov/edr-site/, 2008.

[7] "NVSRAM: A Non-Volatile Static Random Access Memory," http://en.wikipedia.org/wiki/NVSRAM, 2008.

[8] Intel Corp., "Understanding the Flash Translation Layer (FTL) Specification," http://www.intel.com/design/flcomp/applnots/297816.htm, 2008.

[9] H. Dai, M. Neufeld, and R. Han, "ELF: An Efficient Log-Structured Flash File System for Micro Sensor Nodes," *Proc. Second Int'l Conf. Embedded Networked Sensor Systems*, pp. 176-187, 2004.

[10] S. Lin, D. Zeinalipour-Yazti, V. Kalogeraki, D. Gunopulos, and W.A. Najjar, "Efficient Indexing Data Structures for Flash-Based Sensor Devices," *Trans. Storage*, vol. 2, no. 4, pp. 468-503, 2006.

[11] A. Mani, M. Rajashekhar, and P. Levis, "TINX: A Tiny Index Design for Flash Memory on Wireless Sensor Devices," *Proc. Fourth Int'l Conf. Embedded Networked Sensor Systems*, 2006.

[12] D. Woodhouse, *JFFS: The Journalling Flash File System*, Proc. Ottawa Linux Symp., 2001.

[13] Aleph1 Company, "YAFFS: Yet Another Flash File System," http://www.yaffs.net/, 2008.

[14] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *Proc. 13th Symp. Operating System Principles*, pp. 1-15, 1991.

[15] A.B. Bityutskiy, *JFFS3 Design Issues V. 0.32*, draft, http://www.linux-mtd.infradead.org/doc/jffs3.html, Nov. 2005.

[16] Spansion, "Simultaneous Read/Write versus Erase Suspend/Resume," http://www.spansion.com/application_notes/erase_susp_appnote_00_a1_e.pdf, 2008.

[17] "Memory Technology Device (MTD) Subsystem for Linux," http://www.linux-mtd.infradead.org, 2008.