Design of a NAND Flash Memory File System to Improve System Boot Time

Song-Hwa Park*, Tae-Hoon Lee*, and Ki-Dong Chung*

Abstract: NAND flash memory-based embedded systems are becoming increasingly common. These embedded systems have to provide a fast boot time. In this paper, we have designed and proposed a flash file system for embedded systems that require fast booting. By using a Flash Image Area, which keeps the latest flash memory information such as types and status of all blocks, the file system mounting time can be reduced significantly. We have shown by experiments that our file system outperforms YAFFS and RFFS.

Keywords: *Fast Mounting, Flash File System, NAND Flash Memory*

1. Introduction

Embedded systems such as mp3 players, digital cameras and RFID readers have limited resources and should be able to provide an instant start-up time [1]. In these systems, flash memory is typically used as a storage system because of its advantages. It is non-volatile, meaning that it retains data even after power has been turned off and consumes relatively little power. In addition, flash memory offers a fast access time and solid-state shock resistance. These characteristics explain the popularity of flash memory for embedded systems. However, the mounting time of a flash file system takes up a large fraction of the system boot time. The flash mounting time heavily depends on the flash capacity and the amount of stored data. Thus, a flash file system to support fast mounting needs to be developed.

The long mounting time occurs due to the two hardware constraints of the write operation in flash memory [2]. First, it is write-once device, the existing data cannot be overwritten directly. In ordinary writing, it can transit from one state (called the initial state) to another, but it cannot make the reverse transition as exemplified in Figure 1. Bold 0 denotes that it cannot be changed to 1 by writing 0xFF. In order to return to initial state, it needs to perform an initialization operation called block erase operation. This means that if you want to change one bit of data to the initial state in flash memory, you must erase a whole block [3]. Second, an erase operation is performed in a larger granularity (i.e. block) than a write operation (i.e. page). To overcome these constraints, the existing file systems such as JFFS2 [4] and YAFFS [5] allocate memory spaces using the LFS (Log-structured File System) [6] method. In the cases of JFFS2 and YAFFS, when update operations occur, the updated data is written to other space. Therefore, these file systems have to scan the entire flash memory to collect the scattered data because many pieces of a file are scattered all around the flash memory. This means that it takes a long time to mount such file systems and can be a critical issue in selecting a flash file system for embedded systems.

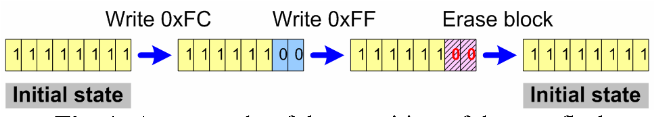


Fig. 1. An example of the transition of data on flash memory

To provide fast mounting, the flash file system called RFFS was proposed. RFFS stores all the block information and the addresses of block information and metadata blocks for using at mounting time. Even though it provides fast mounting, it can be improved further by reducing the amount of data to be scanned when mounting the file system [7].

In this study, we aim to design and implement a flash file system for NAND flash memory-based systems. The proposed flash file system stores the flash memory image which includes the in-memory block status for fast mounting. When unmounting the file system, the flash memory image is written to the fixed location of the flash memory. When mounting the file system, it reads the flash memory image and constructs the block information in RAM. Then, it reads only the metadata blocks using block information, so it can construct the file system quickly.

The rest of this paper is organized as follows. In Section 2, we describe the existing NAND flash file systems. The proposed file system is described in Section 3. The

Manuscript received October 13, 2006; accepted November 24, 2006.
This work was supported by the Regional Research Centers Program (Research Center for Logistics Information Technology), granted by the Korean Ministry of Education & Human Resources Development.
Corresponding Author: Song-Hwa Park
* Dept. of Computer Engineering, Pusan National University, Pusan, Korea (downy25@empal.com, withsoul@melon.cs.pusan.ac.kr, kdchung@pusan.ac.kr)

evaluation results are presented in Section 4, and the conclusion is shown in Section 5.

2. Related Work

In this section, we introduce YAFFS and RFFS.

2.1 YAFFS [5]

YAFFS (Yet Another Flash Filing System) [5] was designed and written by Charles Manning of the company Aleph One. YAFFS is the first file system designed specifically for NAND flash memory.

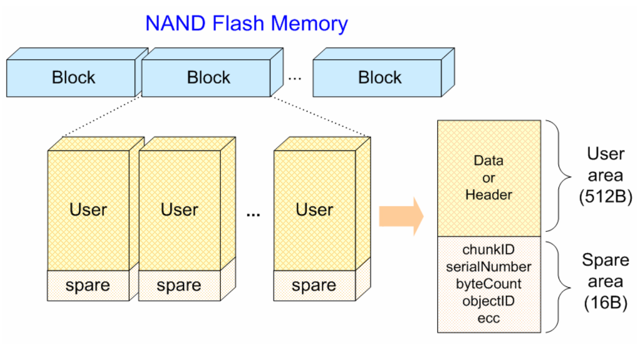


Fig. 2. Flash memory structure of YAFFS

Figure 2 shows the organization of YAFFS. It works on NAND chips that have 512-byte pages and 16-byte spare areas. Data is stored on the NAND flash in chunks. Each chunk is the same size as the user area. It can hold either an object header or the file data. The spare area contains information about the corresponding chunk such as chunkID, serialNumber, byteCount, objectID, ECC and others. A chunkID of zero indicates that the chunk holds an object header which includes the name, size, modified time of the object, and so on. A non-zero value indicates that the chunk contains the data and the value denotes the position of the chunk in the file. File data locations are maintained in a tree structure.

2.2 RFFS [7]

RFFS was proposed to provide fast mounting regardless of the flash memory capacity and amount of stored data. To meet this requirement, we proposed the file system architecture for flash memory shown in Figure 3.

In the case of RFFS, the flash memory is partitioned into two areas, the Location Information Area (referred as LIA) and the General Area (referred as GA), which are managed separately. In particular, the LIA maintains the latest location information. It occupies several groups of blocks and is initially read into the main memory during the mounting. The GA is the remaining area, except for the LIA in the flash memory. All sub-areas such as metadata, data and block information are stored in this area.

Loc_Info, the data structure for location information, is shown on the left-hand side of Figure 3. *Loc_Info* consists of the block_info and meta_data fields. The block_info fields point to the location where the latest block information is written. An array of meta_data fields stores the addresses of the metadata sub-area.

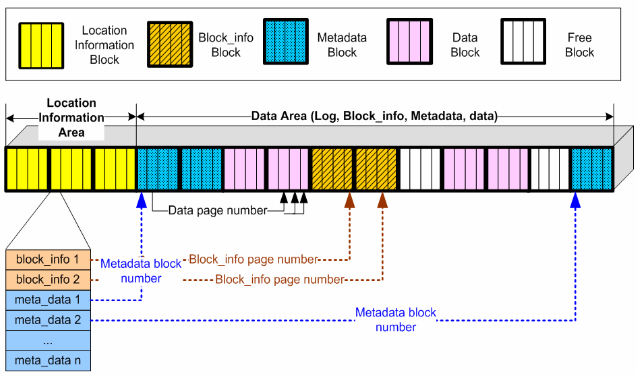


Fig. 3. Flash memory structure of RFFS

GA includes all sub-areas such as metadata, file data and block_info. These sub-areas (except data block) are managed in a segment unit which is composed of several blocks. The metadata sub-area consists of a number of independent segments. We store all metadata for objects such as files, directories, hard links and symbolic links in this area. Unlike the conventional flash file system, such as JFFS2 and YAFFS, RFFS contains file locations in metadata. Since all metadata are stored in metadata sub-area, we can construct the data structures in RAM by only scanning the metadata sub-area during the mounting. Block_info sub-area stores the *Block_Info* data structures that contain the status of all blocks in flash memory. For each block, *Block_Info* keeps the information on the number of pages in use, block status, block type, and so on. RFFS makes use of this information to determine such policies as new block allocation and garbage collection. When unmounting the file system, the latest *Block_Info* data structures are written to flash memory.

Even though RFFS can provide fast mounting, it can be further improved by reducing the amount of data to be read when mounting the file system. There is no need to store all the block information when a part of the blocks is used. It may waste the flash memory space to write all block information and delay the mounting of the file system.

3. Design of a NAND Flash File System

In this section, we describe the NAND flash file system architecture to improve the system boot time.

3.1 On-Flash Data Structures

To satisfy our goal, we propose a new flash file system in flash memory as shown in Figure 4. A metadata of the proposed file system contains the file data or data locations depending on the file size. This can improve the flash memory availability.

In the proposed architecture, the flash memory is partitioned into two areas, Flash Image Area (referred as FIA) and Data Area (referred as DA), which are managed separately. In particular, FIA maintains the latest flash memory image. It occupies several groups of blocks and is first scanned at mounting time. DA is the remaining area, except for the FIA in the flash memory space. In this area, metadata and file data are stored. Let us show the characteristics of each area one by one.

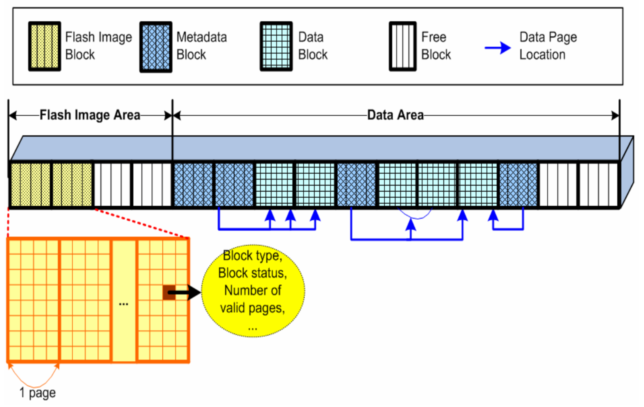


Fig. 4. The proposed flash file system architecture in flash memory

3.1.1 Flash Image Area

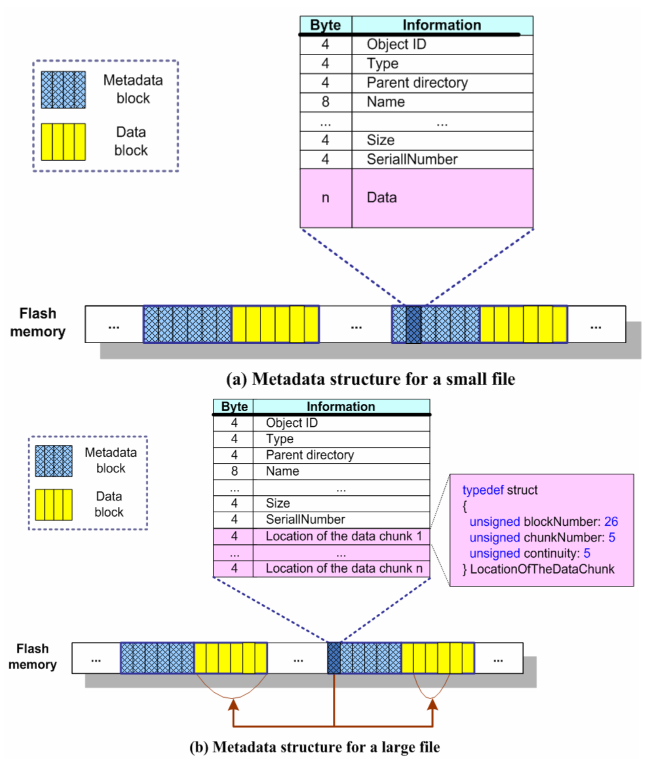
The FIA keeps the latest flash memory information which includes the content type and status of all blocks. The FIA is set to a fixed size and used in a round-robin manner.

The FIA stores the latest flash memory image which consists of the *Block_Info* data structures shown on the left-hand side of Figure 4. As shown in Figure 4, *Block_Info* for each block keeps the block type, block status, number of pages in use, and so forth. When unmounting the file system, the *Block_Info*s of used blocks are written in the FIA. Once the flash memory image has been written, the pages containing the previous flash memory image are invalidated.

3.1.2 Data Area

DA stores metadata and file data. We store all metadata for objects such as files, directories, hard links and symbolic links in this area. Each block has the type information which indicates the stored data type of the block. This means that if a block is allocated for storing metadata, the remaining space can be used only for storing metadata. Therefore, metadata and file data are not stored in the same block.

When a file operation occurs, the space for the metadata is allocated. Figure 5 shows the *Meta_Data* structures of the proposed file system. The structure of the metadata is represented in one of two ways, depending on the data size. For small files, the data in the file is stored in the metadata, as shown in Figure 5 (a). We can store data with metadata when the file size is 320 bytes and less. If there are many small files, we can improve the availability of the flash memory because it uses only one page for storing metadata and file data. For large files, the metadata contains the locations of data pages containing the file data, as shown in Figure 5 (b). Unlike conventional flash file systems such as JFFS2 and YAFFS, we do not need to scan the entire flash memory to collect scattered data since the metadata contains the file data or data locations.



Byte	Information
4	Object ID
4	Type
4	Parent directory
8	Name
...	...
4	Size
4	SeriallNumber
n	Data

(a) Metadata structure for a small file

Byte	Information
4	Object ID
4	Type
4	Parent directory
8	Name
...	...
4	Size
4	SeriallNumber
4	Location of the data chunk 1
...	...
4	Location of the data chunk n

```
typedef struct
{
  unsigned blockNumber: 26
  unsigned chunkNumber: 5
  unsigned continuity: 5
} LocationOfTheDataChunk
```

(b) Metadata structure for a large file

Fig. 5. Management of files using *Meta_Data* structures

3. 2 In-Memory Data Structures

All directories, files, hard links and symbolic links are abstracted to objects. A file system mounting procedure includes the construction of block status and the creation of data structures for objects in RAM. *Object* structures are created for the run-time support of operations on objects and managed by using a list.

3.2.1 Management of Block Information

To manage the block information, arrays of *Block_Status* and block number are used.

The *Block_Status* data structures are created in RAM by using a flash memory image. They contain the information on their corresponding block information and are managed in an array. The index in an array denotes the corresponding block number. We can make use of *Block_Status* to determine such policies as space allocation and garbage collection.

Another array called *UsedBlockNumber* is used for storing the numbers of used blocks. We can store information on the used blocks using this array. When a block is allocated, the block number is inserted into the array.

The *Block_Status* and *UsedBlockNumber* in RAM reflect the changes in block status in the flash memory. Figure 6 shows an example of updating *Block_Status* and *UsedBlockNumber*.

(1) As an application program performs a write operation, we allocate space using *Block_Status*. The status of the 30th block is changed as the pages in the block are allocated for writing. Therefore, the *Block_Status* of the 30th block is updated.

(2) As we allocated the 30th block, the block number is inserted into the tail of *UsedBlockNumber* array.

(3) Write data to the 30th block of flash memory.

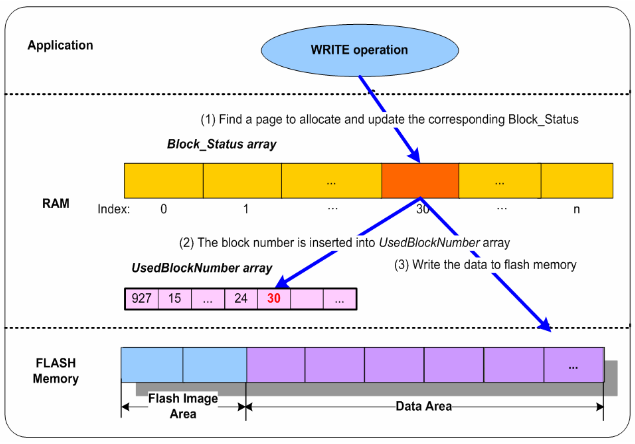

Application
WRITE operation
(1) Find a page to allocate and update the corresponding Block_Status
Block_Status array
RAM
Index: 0 1 ... 30 ... n
(2) The block number is inserted into UsedBlockNumber array
UsedBlockNumber array
(3) Write the data to flash memory
927 15 ... 24 30 ...
FLASH Memory
Flash Image Area
Data Area

Fig. 6. An example of *Block_Status* and *UsedBlockNumber* updates in RAM

3.2.2 Object data structure

All directories, files, hard links and symbolic links are abstracted to *Object* data structures. *Object* data structures are created in RAM for the run-time support of operations on directories, files, hard links or symbolic links. During the mounting, the *Object* data structures are created in RAM by loading metadata. An *Object* knows about its name, type, corresponding metadata location in flash memory, and so on. Modifications to directories, files, hard links or symbolic links are reflected on the *Object* as they occur.

To provide fast run-time support operations on a file, *Object* has to know its data locations. An *Object* of file has a tree structure for storing file data locations. When creating a file, a tree structure is created for the file. The data locations of the file are stored in the tree. The tree is reduced or expanded according to the file size.

3.3 Mounting Procedure

Now, let us explain the mounting process described in Figure 7. First, initialize *Block_Status* array. Then, set *Block_Status* by loading *Block_Info* from the FIA. Also, the block number of the read *Block_Info* is inserted into the *UsedBlockNumber* array. After finishing loading all *Block_Info*, we must determine which block to read. The block number is obtained from the *UsedBlockNumber* array. Then, the *Block_Status* of the block is obtained and the type of the block is checked. If it is a metadata block, we read the block and construct an *Object* in RAM. YAFFS and RFFS mark every newly written page with a serial number that is monotonically increasing. Thus when YAFFS and RFFS scan the flash, they may detect the multiple data pages of one file that have an identical ChunkID. They can choose the latest page by taking the greatest serial number. However, we can choose the latest data by taking the recently read page because we read the metadata block according to an allocated sequence. Furthermore, we do not need to read the file data blocks since the metadata contains the file data or data locations. Therefore, we can reduce mounting time and improve system boot time.

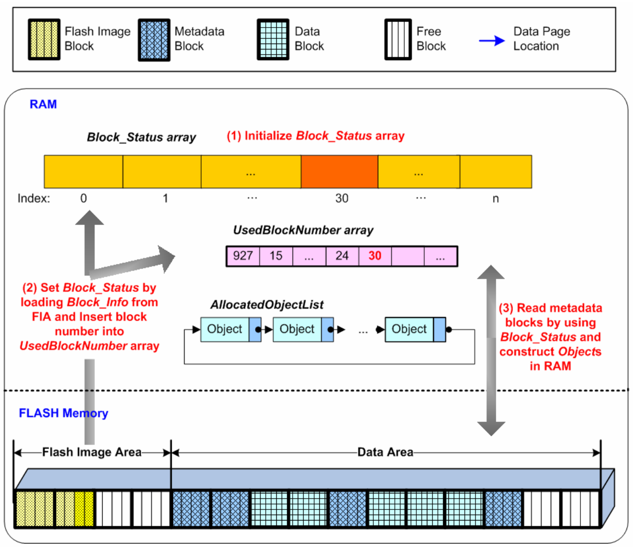


Fig. 7. Mounting procedure of the proposed file system

3.4 Unmounting Procedure

To provide fast mounting, we should store the information required during the mounting at unmounting time. In the case of RFFS, it writes information on locations and all the blocks to the flash memory; as such, it wastes flash memory space.

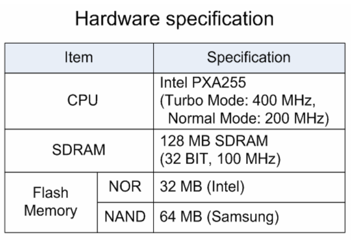
The proposed file system stores the information of the used blocks at unmounting time. We obtain the number of used blocks from *UsedBlockNumber* and the block information from *Block_Status*. Therefore, the amount of written data varies according to flash memory usage.

4. Experimental Results

In this section, we evaluate the mounting time of the proposed file system as compared with YAFFS and RFFS.

4.1 Experiment Environment

We implemented the proposed file system and conducted various experiments using an embedded board running Linux kernel 2.4. We used the PXA255-Pro III board made by Huins. Figure 8 summarizes the PXA255-Pro III board specification. We used a 60-MB Samsung NAND flash memory whose I/O characteristics are shown in Table 1. The block size of the memory is 64 KB and the chunk size is 512 bytes. Since the mounting time of the flash file system depends heavily on data size and flash memory usage, we evaluated the performance by increasing the flash memory usage. For the experiments, we created test data with reference to the write access denoted as in [8]. The average file size is around 22KB, and most files are smaller than 2KB.



Hardware specification

Item		Specification
CPU		Intel PXA255 (Turbo Mode: 400 MHz, Normal Mode: 200 MHz)
SDRAM		128 MB SDRAM (32 BIT, 100 MHz)
Flash Memory	NOR	32 MB (Intel)
	NAND	64 MB (Samsung)

Fig. 8. PXA255-Pro III board specifications

Table 1. I/O Characteristics of NAND flash memory

Operation Type	Read	Write	Erase
Operation Unit	512 B	512 B	16 KB
Speed	15 ㎲	200 ㎲	2 ms

4.2 Mounting Time Comparisons

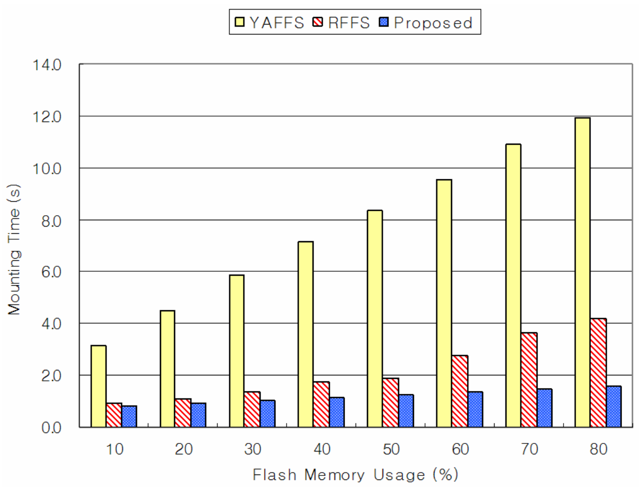


Fig. 9. Mounting time comparison according to flash memory usage

Figure 9 shows the average mounting time of YAFFS, RFFS, and the proposed file system, respectively. We measured the mounting time by increasing the flash memory usage from 10% to 80%. The result shows that the proposed file system exhibits the best performance. However, YAFFS shows the poorest performance compared with the other file systems, because it fully scans the flash memory regardless of flash memory usage. In contrast to YAFFS, RFFS and the proposed file systems do not need to scan the entire flash memory space. Tables 2 and 3 show the numbers of read spares and pages during the mounting: these show that RFFS and the proposed file system read much smaller spares and pages than YAFFS at mounting time. Furthermore, the proposed file system can be mounted by reading smaller spares and pages than RFFS. RFFS improved the mounting time of YAFFS by 65%~76%. Our file system improved the mounting time of YAFFS by 74%~87%.

Table 2. Number of read spares at mounting time

Flash Memory Usage	10%	20%	30%	40%	50%	60%	70%	80%
YAFFS	4,920	8,640	12,360	16,080	19,800	23,520	27,240	31,680
RFFS	509	990	1,471	1,948	2,429	2,910	3,391	3,868
Proposed File System	485	967	1,450	1,932	2,414	2,897	3,379	3,862

Table 3. Number of read pages at mounting time

Flash Memory Usage	10%	20%	30%	40%	50%	60%	70%	80%
YAFFS	3,840	7,680	11,520	15,360	19,200	23,040	26,880	30,720
RFFS	506	986	1,466	1,946	2,426	2,906	3,386	3,866
Proposed File System	484	966	1,449	1,931	2,413	2,896	3,378	3,861

5. Conclusion

In this paper, we designed a new NAND flash file system for embedded systems. To support fast mounting, we divide the flash memory into the Flash Image Area and the Data Area. During the mounting, we read only the flash memory image and metadata blocks. The metadata contains the file data or data locations. Therefore, we can improve the flash memory availability and the mounting time because it does not need to read the data blocks.

We evaluated our proposed file system by experiments. According to the results, we improved the mounting time by 74%~87% in terms of flash usage when compared with YAFFS.

For future work, we are planning to develop an efficient wear-leveling algorithm suitable for the proposed file system. We should also study and develop a journaling mechanism in order to provide file system consistency against sudden system faults.

Reference

[1] T.R. Bird, “Methods to Improve Bootup Time in Linux,” *Proceeding of the Ottawa Linux Symposium (OLS), Sony Electronics*, 2004.

[2] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, “Introduction to Flash Memory”, *Proceeding of the IEEE*, vol. 91, No. 4, pp. 489-502, April 2003.

[3] Understanding the Flash Translation Layer (FTL) specification. Intel: 1997.

[4] D. Woodhouse, “JFFS: The Journaling Flash File System", *Proceeding of the Ottawa Linux Symposium*, RetHat Inc. 2001.

[5] Aleph One Company, “The Yet Another Flash Filing System (YAFFS)”, http:// www.aleph1.co.uk/yaffs/yaffs.html

[6] M.Resenblum and J.K.Ousterhout, "The Design and Implementation of a Log-Structured File System", *ACM Transaction on Computer Systems*, vol.10, pp.26-52, 1992.

[7] S.H.Park, T.H.Lee, K.D.Chung, "A Flash File System to Support Fast Mounting for NAND Flash Memory-Based Embedded Systems", SAMOS 2006, *Lecture Notes in Computer Science*, vol.4017, pp.415~424, 2006.

[8] G. Irlam, “Unix File Size Survey”, http://www.base.com/gordoni/gordoni.html

Song-Hwa Park
Park received a B.S. degree in Computer Engineering from Pusan National University in 2005. She is currently pursuing a M.S. degree as a member of the parallel multimedia lab at Pusan National University. Her research interests are in the area of flash file systems, multimedia and embedded systems.

Tae-Hoon Lee
Lee received B.S. and M.S. degrees in Computer Science from Pusan National University in 2004 and 2006, respectively. He is now undertaking a doctorate course as a member of the parallel multimedia lab at Pusan National University. His research interests include embedded systems, wireless networks, and file systems.

Ki-Dong Chung
He received a B.S. degree from Seoul National University in 1973, and M.S. and Ph.D. degrees in Computer Science from the same university, in 1975 and 1986. Since 1978, he has been a Professor of omputer science at Pusan National University, Pusan, Korea. His research interests include VOD (Video On Demand), multimedia systems, and mobile multimedia communication.