Fast Mounting and Recovery for NAND Flash Memory Based Embedded Systems*

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

Dept. of Computer Engineering, Pusan National University,
Kumjeong-Ku, Busan 609-735, Korea
{downy25, rider79, withsoul}@melon.cs.pusan.ac.kr,
kdchung@pusan.ac.kr
http://apple.cs.pusan.ac.kr

Abstract. Even though its advantages such as non-volatility, fast write access time and solid-state shock resistance, NAND flash memory suffers from out-place-update, limited erase cycles, and page-based I/O operations. How to provide fast mounting and consistency of file system and data for flash memory file systems has become important research topics in recent years. In this paper, we design and propose a new flash memory file system called RFFS (Reliable Flash File System), which targets NAND flash memory based embedded systems that require fast mounting and fault tolerant file system. We have experimented on the performance of RFFS and the results showed that RFFS could improve the mounting time by 65%–76% compared with YAFFS. Also RFFS improved mounting time after sudden system faults by 89%–92% compared with JFFS2.

1 Introduction

Embedded systems such as mp3 player, digital camera and RFID reader should be able to provide an instant start-up time [1]. In these systems, flash memory is typically used as storage system because of its attractive features. It is non-volatile, meaning that it retains data in the chip even after power is turned off and consumes relatively little power. In addition, flash memory offers fast access and solid-state shock resistance. These characteristics explain the popularity of flash memory for embedded systems as storage.

There are two major type of flash memory according to the gate type and structure of the memory cell: NOR flash and NAND flash. As NOR flash memory supports byte-unit I/O and provides fast read speed, it is usually used as a code storage medium. Conversely, NAND flash memory supports only page-based I/O and provides fast write speed, so it is widely used as a large-scale data storage medium [2].

Despite the advantages of NAND flash memory, it has several characteristics that make straightforward replacement of existing storage media difficult. First, it suffers from out-place-update. In ordinary writing, it can transit from one state

* 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.

X. Zhou et al. (Eds.): EUC Workshops 2006, LNCS 4097, pp. 710–720, 2006.

(called initial state) to another state, but it can't make the reverse transition. In order to return to initial state, it needs to perform initialization operation called block erase operation. It 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, blocks have limited endurance. Therefore, erase operation is avoided for longer flash memory lifetimes. Last, the erase operation must be done evenly to all blocks to avoid wearing out of specific blocks to affect the usefulness of the entire flash memory. This is usually named as wear leveling or cycle leveling.

Because the conventional file systems cannot be applied directly to flash memory due to above mentioned characteristics, new flash file systems such as JFFS2 [4] and YAFFS [5] were developed. The JFFS2 file system is a version of a journaling file system based on LFS [6]. Files are broken into several smaller nodes, which contain the actual data. Every time updated data is written to flash memory, a new node is created and is written to another location in flash. Therefore, all nodes must still be scanned at mounting time. This is slow and consumes enormous memory. YAFFS is the first file system that is designed specifically for NAND flash. It outperforms JFFS2 with respect to mounting time and memory consumption. However, when mounting YAFFS, it scans the spare areas of every block to check validation of data. As with JFFS2, YAFFS also has a long mounting time problem. Also YAFFS can not guarantee the consistency of file contents even if it maintains file system consistency.

Since flash chip capacity is doubling every year, the mounting time will soon become the most dominant reason of the delay of system start-up time [7]. The goal of this research is to design and implement a new fast and reliable flash file system called RFFS, which minimizes mounting time and provides consistency of file system and contents of files at abrupt failures for such embedded systems.

This paper is organized as follows. In Section 2, we describe JFFS and YAFFS, the NAND flash file systems, respectively. In Section 3, we present our proposed file system to support fast mounting and reliability. The evaluation results are presented in Section 4, and the conclusion is shown in Section 5.

2 Flash Memory File Systems

In this section, we introduce the flash file systems for NAND flash memory, JFFS2 and YAFFS.

2.1 JFFS2

The JFFS file system was originally developed for small NOR ($\leq$32MB) flash memory by Axix Communications in Sweden. David Woodhouse and others developed JFFS2 by improving JFFS. JFFS2 addresses most of the issues of flash file systems by improving read/write time and providing compression, wear-leveling and journaling [8]. NAND support was added later to JFFS2.

JFFS2 is simply a list of nodes and log entries. The node contains actual file data and the log entry holds information about a file. All logs should be scanned to determine how to create files at mounting time. JFFS2 uses flash

memory in a round-robin manner. Blocks are consecutively written until the end of flash is reached, then starts at the beginning again. JFFS2 includes a garbage collection thread that combines and copies valid nodes to another block, then erases partially used blocks. The process of sequentially erasing and writing flash blocks provides wear-leveling, as it distributes the flash writes over the entire flash device.

Even though JFFS2 solves most issues of flash memory file systems, it still has some problems because the arbitrary size of journaling nodes causes a fragmentation of pages on NAND flash memory. Especially, JFFS2 has faced serious problems as NAND flash chips become larger. First, the mounting time becomes too long. Second, the memory consumption becomes enormous. Third, the access to large files takes a long time. To overcome these problems, the JFFS3 [9] draft was issued.

2.2 YAFFS

YAFFS (Yet Another Flash Filing System) was designed and written by Charles Manning of the company Aleph One. YAFFS is the first file system designed specifically for small NAND ($\leq$32MB) flash memory which has 512 byte pages and 16 byte spare areas. Data is stored on NAND flash in chunks. Each chunk is the same size as a NAND flash page. Chunk can hold either an object header or a chunk of file data. The 16 byte spare area contains the information about the corresponding chunk such as chunkID, serialNumber, byteCount, objectID and ECC (Error Correction Code) and others. A chunkID of zero indicates that this 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 file data and the value means the position of chunk in the file [10].

When a chunk is no longer valid, YAFFS marks a particular byte in the spare area as dirty. When an entire block is marked as dirty, YAFFS can erase the block and reclaim the space. If free space on the device is low, YAFFS may need to choose a block that has some number of dirty pages. In this case, the valid pages are moved to a new block and the old pages are marked as dirty. YAFFS writes updated chunk with a serial number that is monotonically increasing. Thereby when YAFFS detects multiple chunks that have identical ObjectID and ChunkID, it can choose the latest chunk by taking the greatest serial number at mounting time. However, in the case of sudden system faults during the write operation, YAFFS can't guarantee the consistency of file data. Since only some of chunks are rewritten with new serial number due to system failure. Accordingly, the new reliable flash file system that provides almost constant mounting time regardless of flash memory size is required.

3 A Design of RFFS

In this section, we describe the RFFS architecture for flash memory and data structures in RAM. Additionally, we present a logging mechanism for providing the file system reliability and fast recovery technique.

3.1 File System Architecture in Flash Memory

RFFS aims to provide fast mounting and file data consistency regardless of system faults. To satisfy these requirements, we designed the file system architecture in flash memory as shown in Fig.1. JFFS2 and YAFFS spread metadata and file data all around flash memory. This scheme causes long mounting time. Therefore, keeping the locations of the related data is the key to support fast mounting.

In the proposed architecture, the flash memory is partitioned into three areas and managed separately, Location Information Area (LIA), Location Information Backup Area (LIBA) and Data Area (DA). LIA occupies several groups of blocks (referred as segment) and is firstly scanned at mounting time. As it maintains the latest location information, LIA is the critical space of RFFS. If read fault, one of the flash memory characteristics, occurs when reading the data in LIA, RFFS must scan the whole flash memory. To solve this problem, we store the copy of location information at LIBA. DA is the remaining area except LIA and LIBA in flash memory space. In this area, all types of sub-areas such as log, block_info, metadata and data are stored. Let us show the characteristics of each sub-area one by one.

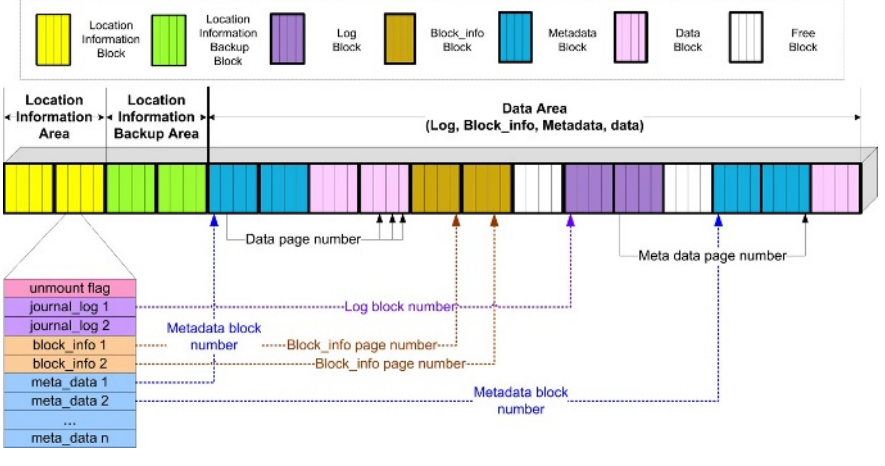

Fig. 1. RFFS architecture in a flash memory system

Location Information Area. LIA keeps the latest location information for sub-areas except data. RFFS can improve the mounting time by making use of location information.

Loc_Info, the data structure for location information, is described in the left side of Fig.1. It is a page-sized data structure due to the limitation of NAND flash memory I/O unit. *Loc_Info* consists of unmount_flag, journal_log, block_info and meta_data fields. Except the unmount_flag, all fields point to sub-areas which contain the real data. The unmount_flag and the journal_log are used to support

file system reliability. The block_info fields point to the block_info sub-area, which keeps the latest flash memory block status. An Array of meta_data field stores the addresses of the metadata sub-areas. The number of indices in the array limits the maximum number of files in the file system. Now, the maximum number of files in RFFS is 15,842 and we can extend this through bit optimization of each field in the *Loc_Info*.

Location Information Backup Area. Read faults such as one bit read errors or read failures can be occurred during the read operation. One bit read errors affect data reliability by changing in one bit of the read bits. Moreover, flash memory can cause read failures meaning that we can't read any data in a page [13]. In RFFS, LIA is the most important area since it contains the locations of sub-areas. If read faults occur during reading the LIA, RFFS may be mounted to abnormal state or fail to be mounted. Therefore, we should deal with read faults since read faults occur sporadically in flash memory.

In case of one bit read errors, RFFS corrects it using ECC (Error Correction Code) stored in spare area. However, if RFFS fails to read a page in LIA, it must scan the entire flash memory to mount file system because ECC can correct only one bit errors. Therefore, RFFS stores the duplicated location information at LIBA against page read failures. Every time the location information is written to LIA, it also written to LIBA. The location information stored in LIBA can be used when RFFS fails to read a page in LIA.

Data Area. DA includes all sub-areas such as log, metadata, file data and block information. These sub-areas except file data are managed based on segment unit and the valid sub-areas are excluded from garbage collection. Let us show the characteristics of each sub-area.

First, the log sub-area consists of two segments as shown Fig. 2. If one of them is fully invalidated, the other is used alternately and the invalid log segment is collected by garbage collector. The log sub-area contains logs which record write operations to recover file system when sudden system faults occur. When a write operation occurs, related metadata address, serial number and commit flag are stored in a log. A log data becomes invalid when transaction is completed.

Second, the metadata sub-area consists of a number of independent segments. The maximum number of metadata is limited by the meta_data array of *Loc_Info* in LIA. RFFS stores all metadata for objects such as files, directories, hard links

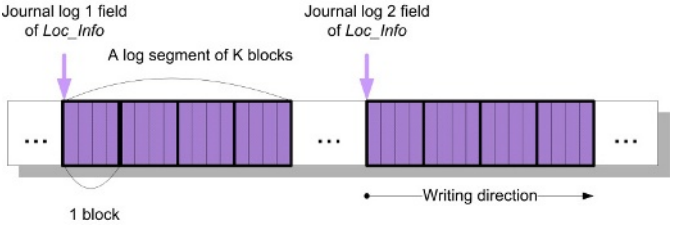

Fig. 2. A log segment and write operation

and symbolic links in this area. Fig. 3 shows the *Meta_Data* structures of RFFS. Unlike the conventional flash file system such as JFFS2 and YAFFS, RFFS contains file data or file location information in metadata. If the file data is small, the data is stored together with metadata in a page. Otherwise, the file data is stored in data blocks and its locations are stored in metadata. For large files, RFFS may request several metadata pages. The several metadata pages are managed by using the inverse linked list.

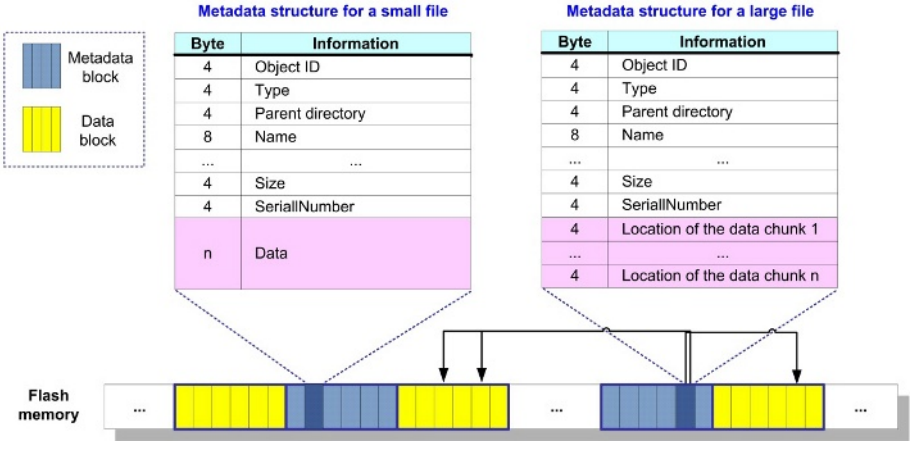

Fig. 3. Management of a file using the Meta_Data structure

Third, the block_info sub-area stores the *Block_Info* data structures that contain the status of all blocks in flash memory and the newly updated block status. For each block, *Block_Info* keeps the number of pages in use, block status, block type, bad block flag and etc., as is shown in Fig. 4. RFFS makes use of this information to determine policies such as space allocation and garbage collection. Although RFFS actually manages the *Block_Info* data structures in RAM, RFFS writes updated *Block_Info*s to flash memory periodically to provide consistency between the real flash memory block status and the corresponding *Block_Info* after sudden system faults. The whole *Block_Info*s are stored in block_info sub-area at unmounting time. Once the whole *Block_Info* structures are written to flash memory, the pages containing the previous *Block_Info* structures are invalidated. The management of this sub-area will be described in section 3.2.

3.2 Management of Data Structures in RAM

A file system mounting process includes construction of block status and creation of data structures for objects in RAM. All directories, files, hard links and symbolic links are abstracted to objects. *Object* structures are created for runtime support of operations on objects and RFFS manages the *object* structures using linked list.

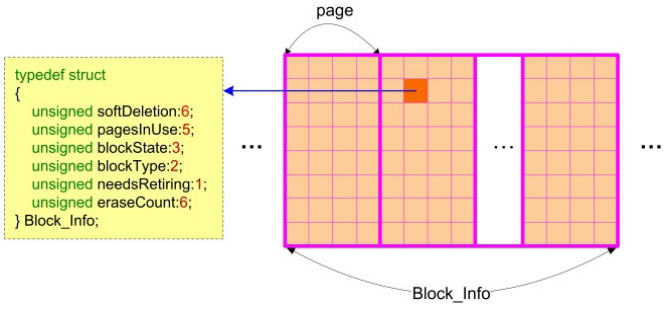

Fig. 4. Block_Info data structure containing status of all blocks in a flash memory

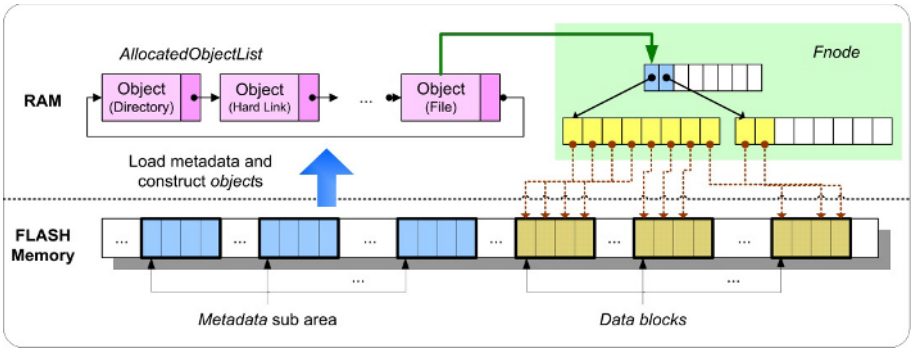

Fig. 5. Object management in RAM

Object Data Structure. RFFS manages the *object* structures using linked list in RAM as is shown in Fig. 5. An *object* structure knows about its corresponding metadata location, object ID, priority and so on. Modifications to the directories, files, hard links or symbolic links are reflected on *object*s in RAM as they occur.

Object structure for a file knows about its data locations. For a file, the *Fnode* data structures are created for managing its data locations. *Fnode* structures form a tree structure that speeds up the search for data chunks in a file. Depending on where it is in the tree, each *Fnode* holds the different information. If it is at the lowest level, then it points to the data locations. Otherwise, it points to lower-level *Fnode*s. Depending on file size, the tree can be reduced or expanded.

Block_Info Data Structure. During the mounting, *Block_Info* data structures are created in RAM. *Block_Info* structures contain the status of all blocks in a flash memory. The *Block_Info* structures reflect the changes on block status. RFFS makes use of block information to determine policies such as space allocation and garbage collection. The Fig. 6 shows an example of *Block_Info* update.

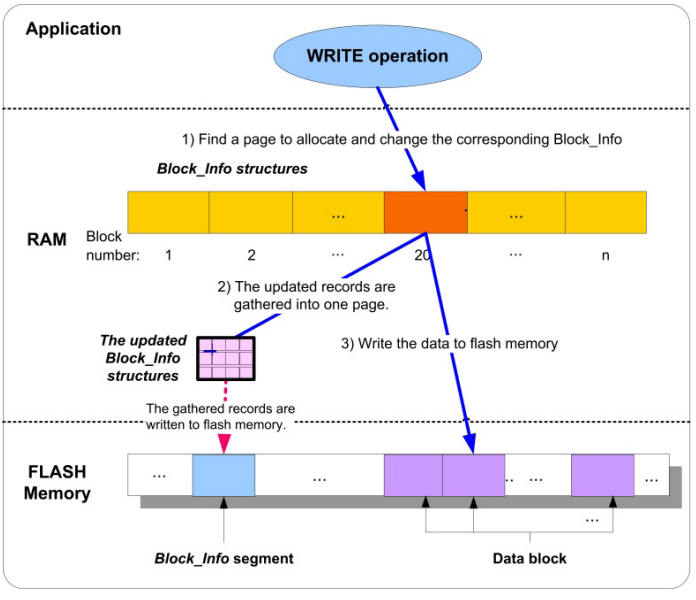

Fig. 6. An example of Block_Info update in RAM

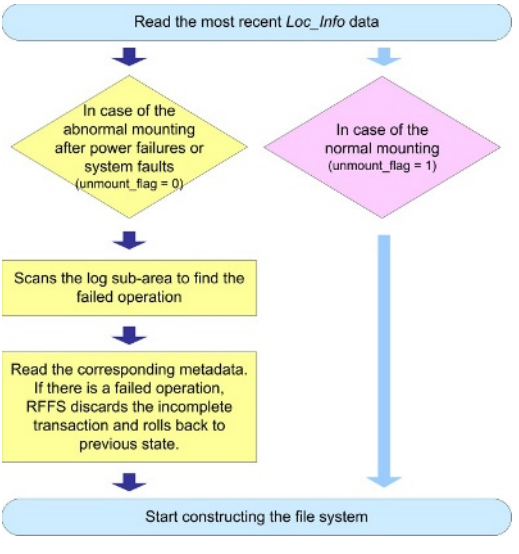

Fig. 7. The proposed fast recovery technique

1. As an application program performs a write operation, RFFS allocates space using the *Block_Info* in RAM.
2. The 20th block status is changed as the pages in the block are allocated for a write operation. The *Block_Info* of the 20th block is updated to reflect the change.

3. The updated records are gathered into one page.
4. The gathered records are written into a free page of block_info sub-area.

3.3 Fast Recovery Technique

Now, let us explain the recovery process described in Fig. 7. If unmount_flag of *Loc_Info* is set to 0, it means that an abrupt failure occurred before completing a transaction. In this case, RFFS scans the most recently used metadata segment to find the failed operation. RFFS discards the incomplete transaction and rolls back to previous state. By exploiting this recovery procedure, RFFS can maintain the consistency of file system and file data and provide fast recovery.

4 Experimental Results

In this section, we show the performance of RFFS through experiments.

4.1 Experimental Environment

We implemented RFFS and experimented using an embedded board. We used PXA255-Pro III board made by Huins. Fig. 8 summarizes the PXA255-Pro III board specification and we used 64MB Samsung NAND flash memory. The block size of the memory is 64KB and the page size is 512B. For experiments, we created test data with reference to write access denoted as in [14].

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 specification

4.2 Mounting Time Performance

Fig. 9 shows the average mounting time of YAFFS and RFFS. We measured the mounting time by increasing the flash memory usage from 10% to 80% on 64MB flash memory. The results show that the mounting time for YAFFS is uniformly high. This is because YAFFS fully scans the flash memory space regardless of flash memory usage to construct the data structures in RAM. In contrast to YAFFS, mounting time of RFFS is in proportion to the flash memory usage due to the amount of block_info, log and metadata sub-areas. So we improved the mounting time of YAFFS by 65%–76%. According to this result, we know that

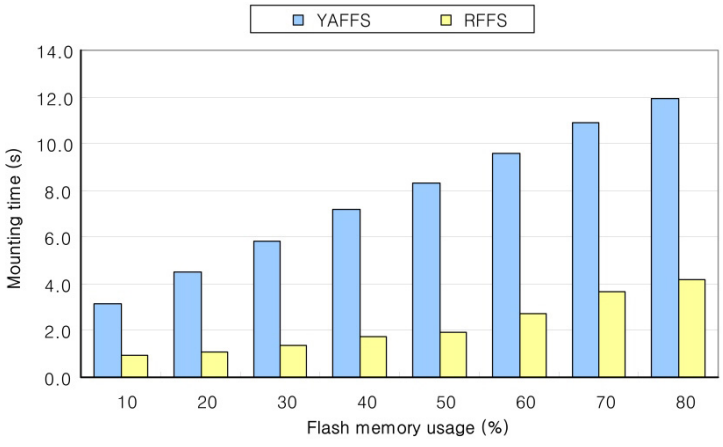

Fig. 9. Mounting time comparison on flash memory usage

it is required to optimize the data structures in DA to improve the mounting time irregardless of flash usage.

4.3 Recovery Performance

Fig. 10 shows the average mounting time of JFFS2 and RFFS after sudden system faults. We compared performances with JFFS2 on 20MB flash memory because YAFFS does not provide the journaling function. The result shows that the mounting time for JFFS2 is uniformly high. This is because JFFS2 fully scans the whole flash memory to check the consistency of file system and construct the data structures in RAM. In contrast to JFFS2, RFFS can maintain the consistency of file system by checking log data regardless of flash memory usage. RFFS improved the mounting time of JFFS2 by 89%–92%.

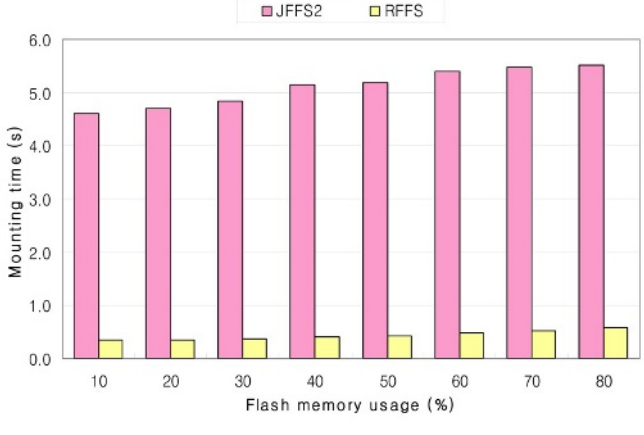

Fig. 10. Mounting time comparison after sudden system faults

5 Conclusion

In this paper, we have designed and implemented a new NAND flash file system called RFFS, which provides fast mounting and consistency for file system and file data. To support fast mounting, RFFS divides flash memory space into Location Information Area, Location Information Backup Area and Data Area. LIA is fixed in size and its location. It stores *Loc_Info* which includes the addresses of sub-areas for important file system data except file data. *Loc_Info* is also stored in LIBA against read faults. During the mounting, RFFS can construct the data structures in RAM using the location information. DA includes actual data for file system such as log, metadata, file data and block information. RFFS improves the mounting time by 65%–76% compared with YAFFS.

We also proposed a fast recovery technique to support the consistency of file system and file data. According to experiment results, RFFS improved the mounting time after sudden system faults by 89%–92% compared with JFFS2.

We are currently developing an effective garbage collector for RFFS. Also we are planning to optimize the data structures for log, metadata and block information to improve the mounting time.

References

1. T.R. Bird: Methods to Improve Bootup Time in Linux. In Proc. of the Ottawa Linux Symposium (OLS). Sony Electronics (2004).
2. Two Technologies Compared: NOR vs. NAND. www.m-sys.com/NR/rdonlyres/24795A9E-16F9-404A-857C-C1DE21986D28/229/NOR_vs_NAND5.pdf
3. Naritaka Ishizumi, Keizo Saisho, and Akira Fukuda: A Design of Flash Memory File System for Embedded Systems: Systems and Computers in Japan Vol.35. No.1. (2004) pp.90-99
4. David Woodhouse: JFFS: The Journaling Flash File System. Technical Paper of RedHat inc. (2001)
5. YAFFS Spec. 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. (1992) pp.26–52
7. Samsung Electronics: Advantages of SLC NAND Flash Memory. http://www.samsungelectronics.com/.
8. Flash Filesystems for Embedded Linux Systems. http://linuxjournal.com/node/4678/.
9. JFFS3 Design Issue. http://www.linux-mtd.infradead.org/tech/JFFS3design/
10. YAFFS. http://en.wikipedia.org/wiki/YAFFS
11. Richard Menedetter: Journaling Filesystems for Linux.
12. Margo Seltzer, Keith Bostic: An Implementation of a Log-Structured File System for UNIX: Winter USENIX (1993) pp.25–29 12.
13. White Paper: Implementing MLC NAND Flash for Cost-Effective, High-Capacity Memory: M-Systems (2003)
14. G. Irlam: Unix File Size Survey. http://www.base.com/gordoni/gordoni.html.