Efficient Metadata Management for Flash File Systems*

Jaegeuk Kim, Heeseung Jo, Hyotaek Shim, Jin-Soo Kim, and Seungryoul Maeng
Computer Science Department
Korea Advanced Institute of Science and Technology (KAIST)
373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, Republic of Korea
{jgkim, heesn, htsim}@camars.kaist.ac.kr {jinsoo, maeng}@cs.kaist.ac.kr

Abstract

NAND flash memory becomes one of the most popular storage for portable embedded systems. Although many flash-aware file systems, such as JFFS2 and YAFFS2, were proposed, the large memory consumption and the long mount delay have been serious obstacles for large-capacity NAND flash memory.

In this paper, we present a new flash-aware file system called DFFS (Direct Flash File System) which fetches only the needed metadata on demand from flash memory. In addition, DFFS employs two novel metadata management schemes, inode embedding scheme and hybrid inode indexing scheme, to improve the performance of metadata operations. Comprehensive evaluation results using microbenchmark, postmark, and Linux kernel compilation trace, show that DFFS has comparable performance to JFFS2 and YAFFS2, while achieving a small memory footprint and instant mount time.

1. Introduction

The demand for storage capacity has been increasing exponentially due to the recent proliferation of multimedia contents. In the meantime, NAND flash memory becomes one of the most popular storage media for portable embedded systems such as MP3 players, cellular phones, PDAs (personal digital assistants), PMPs (portable media players), and in-car navigation systems. Although NAND flash memory is still expensive than magnetic disk, its distinctive features, such as small and lightweight form factor, solid-state reliability, and low power consumption, have made NAND flash memory the standard choice for portable data storage [4, 10]. However, it is necessary to be aware of the following unique characteristics in dealing with NAND flash memory.

- **Out-of-place updates**: NAND flash memory is a storage medium which does not allow in-place updates. In other words, the previous data cannot be overwritten directly at the same location without being erased first. To make the problem worse, an erase operation should be performed on a larger area containing the original data, not on the particular data selectively. For coping with the *erase-before-write* characteristics of NAND flash memory, many flash-based softwares employ a strategy in which the new data are written into an empty space and the old data are invalidated. Due to this out-of-place update scheme, the physical location of data changes whenever it is overwritten.

- **Asymmetric operation latency**: In NAND flash memory, the read latency is much shorter than the write latency by about a factor of 8. Since a write operation sometimes involves an erase operation, it may suffer from long, nondeterministic delays [6].

To overcome these limitations, two major approaches have been proposed for NAND flash memory. One approach is to use Flash Translation Layer (FTL) [7, 8] and the other is flash-aware file systems [2, 3, 5, 15]. FTL is usually employed between operating system and flash memory. The main role of FTL is to emulate the functionality of block device with flash memory by hiding the erase-before-write characteristics as much as possible. Once FTL is available on top of NAND flash memory, any disk-based file system can be used. However, since FTL is operating at the block device level, FTL does not have any access to file system-level information and this may limit the file system performance.

On the other hand, several flash-aware file systems, such as JFFS2 [15], YAFFS2 [2], ELF [3], and TFFS [5], have been developed to simplify the file system design without being in need of FTL and to extract maximum performance

*This work was supported by the IT R&D program of MIC/IITA. [2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software] out of flash memory. Currently, JFFS2 or YAFFS2 serves as one of the most widely used general-purpose flash file systems in an embedded environment. However, the large memory consumption and the long mount delay shown in these file systems have been severely criticized in the previous literature [5, 16].

In this paper, we propose a new metadata management scheme for flash-aware file systems which offers a small memory footprint and instant mount time. In addition, we propose a new directory structure and inode indexing scheme to match the directory operation performance of JFFS2 or YAFFS2. The prototype file system called DFFS (Direct Flash File System) has been developed to demonstrate the effectiveness of the proposed scheme.

The rest of this paper is organized as follows. Section 2 overviews two representative flash file systems, JFFS2 and YAFFS2, and describes the motivation of this paper. Section 3 presents design issues for DFFS. Section 4 shows the efficiency of the proposed metadata management scheme through experimental evaluations. Finally, we conclude in Section 5.

2. Background and Motivation

2.1. JFFS2

JFFS2 is a log-structured file system designed for flash memories [15]. The basic unit of JFFS2 is a node in which variable-sized data and metadata of the file system are stored. Each node in JFFS2 maintains metadata for a given file such as the physical address, its length, and pointers to the next nodes which belong to the same file. Using these metadata, JFFS2 constructs in-memory data structures which link the whole directory tree of the file system. This design was tolerable since JFFS2 is originally targeted for a small flash memory. However, as the capacity of flash memory increases, the large memory footprint of JFFS2, mainly caused by keeping the whole directory structure in memory, becomes a severe problem. The memory footprint is usually proportional to the number of nodes, thus the more data the file system has, the more memory is required. For example, a JFFS2 file system containing 128MB data in 512-byte nodes requires more than 4MB of memory.

Another problem of JFFS2 is the long mount delay. When the file system is mounted, JFFS2 scans the entire flash memory media to check CRC in every node and build the directory structure. The mount time takes from several to tens of seconds depending on the number of nodes in the file system.

2.2. YAFFS2

YAFFS2 is another variant of log-structured file system [2]. The structure of YAFFS2 is similar to that of the original JFFS2. The main difference is that node header information is moved to the NAND spare area and every data unit, called chunk, has the same size as NAND pages to efficiently utilize NAND flash memory. Similar to JFFS2, YAFFS2 keeps data structures in memory for each chunk to identify the physical location of the chunk on flash memory. It also maintains the full directory structure in main memory since the chunk representing a directory entry has no information about its children. In order to build these in-memory data structures, YAFFS2 scans all the spare areas across the whole NAND flash memory. Therefore, YAFFS2 faces the same problems as JFFS2.

2.3. Motivation

As described in the previous subsections, JFFS2 and YAFFS2 consume a lot of memory to retain the complete directory structure in memory. The mount even fails if there is not enough memory. One way to reduce the memory footprint is to fetch only the needed metadata on demand from flash memory as is done in most disk-based file systems. The fetched metadata are cached in memory for a while to accelerate the subsequent accesses to the same metadata, and can be discarded later under memory pressure.

This scheme, however, results in long latency during file system operations. It is reported that metadata operations occupy up to 70% of file system operations [11]. Compared to JFFS2 and YAFFS2 which maintain most of metadata in memory, the on-demand metadata fetching may incur sizeable overhead. Thus, it is important to minimize performance degradation during metadata operations.

This paper investigates two novel metadata management schemes to improve the performance of metadata operations. First, we speed up the directory lookup operation by storing the directory inode and its associated directory entries in one flash page. The traditional directory lookup operation repeats the following three steps until it reaches the end of pathname: (1) read the inode of the current directory, (2) read the contents (directory entries) of the current directory, and (3) search for the target child name in directory entries and obtain the corresponding inode number. In most file systems, inodes are kept separately from its data. Therefore, two flash read operations, one for the directory inode and the other for its contents, are needed whenever a directory lookup operation is performed. Instead, we embed directory entries with the directory inode in the same flash memory page so that the directory lookup operation can be carried out by just one flash read operation. This is feasible as the current NAND flash memory has a rather large page

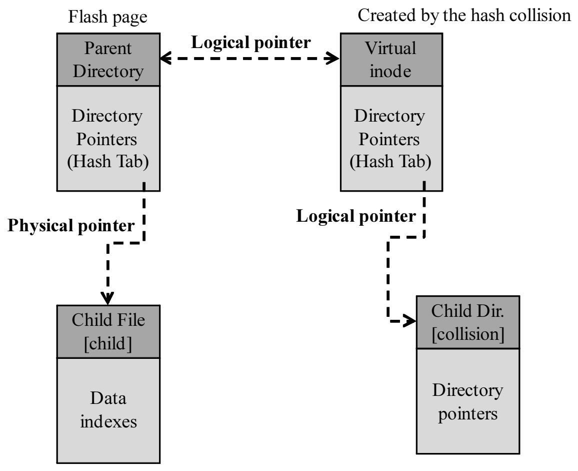

Figure 1. DFFS directory structure

size ranging from 2KB to 4KB.

The other scheme we consider is the hybrid inode indexing structure. In disk-based file systems, the inode of a file can be freely updated whenever there is a change in the file. In flash memory, however, the updated inode cannot be written into the previous location due to the erase-before-write characteristics. Hence, inodes will float over the flash memory area and the file system should be able to locate a file's latest version of inode. JFFS2 and YAFFS2 keep such information in memory and that is why they require so much memory. The log-structured file system (LFS) [12] solves this problem by introducing an *inode map*, which is basically a mapping table from the inode number to the current copy of the inode. Our hybrid inode indexing scheme is similar to LFS except that the *physical* location of a regular file's inode is directly specified in the corresponding directory entry. Only the directory inodes are accessible through the mapping table. This scheme reduces the size of the mapping table significantly, since regular files outnumber directories in most cases.

3. Direct Flash File System (DFFS)

3.1. Directory Structure

In this paper, we propose *inode embedding scheme* to enhance the directory structure by reflecting that the read/write unit of NAND flash memory is a page[1]. As shown in Figure 1, the main idea is to fit a directory inode and the associated directory entries in one flash page. By embedding an inode into its contents, the directory lookup operation can be completed by reading one flash memory page. Likewise, when a file is created or deleted, this scheme requires only one flash write operation by updating the parent inode and its directory entries together.

The overall directory structure of DFFS can be implemented using either a balanced tree or a hash table. Although the B^+-tree-like structure works reasonably well under circumstances that numerous files are created and deleted dynamically [9], it may show extra update cost in flash memory caused by maintaining indirect indexing pages during split and merge operations. For simplicity, DFFS adopts a hash table which performs the average lookup operation in $O(1)$ and has low update cost.

To use the hash table efficiently, hash collisions should be handled carefully. When a hash collision occurs, DFFS allocates another virtual inode holding hash buckets and makes a doubly linked list with the original directory inode as shown in Figure 1. In DFFS, hash collisions occur in proportion to the number of files in a directory.

3.2 Inode Indexing Structure

There are two inode indexing schemes used in flash-aware file systems. One is a *logical (indirect)* indexing scheme which refers to a mapping table entry to acquire the physical location of an inode. The other is a *physical (direct)* indexing scheme which indicates the page location of an inode directly. Although the logical indexing scheme includes mapping table access overhead, it can easily update the physical location of an inode without affecting the parent directory entry. On the other hand, while the physical indexing scheme has low access latency, the parent directory entry should be also updated whenever the location of the child inode changes. This necessitates another update in the grandparent directory and, eventually, updates are propagated to the root directory. Such recursive index updates caused by out-of-place update is called a *wandering tree problem* [1].

DFFS introduces *hybrid indexing scheme* to mitigate the wandering tree problem, to reduce the mapping table size, and to enhance the file access latency. DFFS has three indexing pointers: (1) directory-to-directory, (2) directory-to-file, and (3) hash-collision pointers, as illustrated in Figure 1. Considering frequent updates of inodes, DFFS adopts the logical indexing scheme between directory pointers. For the directory-to-file pointers, DFFS takes the physical indexing scheme by substituting the inode number in the directory entry with the physical location of the inode. This makes it possible not only to access all the children directly, but to reduce the size of the mapping table required for the logical indexing scheme. In addition, the file system does not have to manage free inode numbers and inode locations for regular files. Finally, DFFS simply uses the logical indexing scheme for hash-collision pointers as the hash colli-

[1]In the current NAND flash memory technologies, the page size is either 2KB (for SLC NAND) or 4KB (for MLC NAND).

sion will be rare.

The mapping table for logical pointers is organized in a linear array and the total size is designed not to exceed over the size of one erase block. For SLC NAND flash memory, the size of an erase block is 128KB which can provide up to 32K entries. Since the number of directories in default Linux installation is less than 20K, we believe the size of one erase block is enough in an embedded environment.

3.3 Directory Operations

This subsection explains the internal details of three representative directory operations: open, create, and delete.

During the open operation, the pathname resolution is handled by the VFS (Virtual File System) layer. All DFFS needs to do is to return the inode that matches the pathname component in the parent directory. First, DFFS reads the directory entries embedded into the parent inode. If there is no hash collision, DFFS locates the child inode using the hash value. In case of hash collision, DFFS follows each of the virtual inodes. To avoid visiting all the collided children inside virtual inodes, DFFS uses an extra small hash table in a directory inode which occupies one byte per a bucket. Consequently, DFFS makes use of two hash tables in each directory inode for better open operation performance.

The create operation can be classified into two cases: one for a regular file and the other for a directory file. In the former case, DFFS allocates one physical index and then inserts it to the corresponding bucket of the parent directory. In the latter case, DFFS inserts the logical index to the parent directory. If a hash collision occurs during this process, a new virtual inode is allocated and linked with the parent inode.

When a file or a directory is deleted, DFFS simply invalidates the parent inode bucket. If it is the last inode in the virtual inode, the virtual inode itself is invalidated.

4 Performance Evaluation

4.1 Evaluation Methodology

We have implemented DFFS on top of Memory Technology Device (MTD) in Linux kernel 2.6.14. The total 256MB of SLC NAND flash memory is emulated by the MTD NAND flash simulator called NANDSIM. There are 2048 erase blocks and each erase block contains 128 pages, where each page is 2KB in size [13].

The general architecture of DFFS is based on the log-structured file system [12, 14]. Whenever it needs a free page for new metadata, DFFS allocates a physical page from the last log pointer. The popular SHA-1 algorithm is used as a hash function which generates a pseudo-random number from the given string.

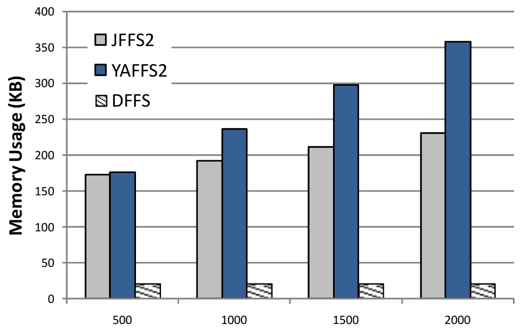

Figure 2. Memory consumption by changing the number of total files

The performance of DFFS has been evaluated using three different workloads. First, we have synthesized a microbenchmark to characterize the basic performance of metadata operations. The microbenchmark constructs a simple directory tree and then measures the average time to process several random operations. Second, we show the result for Postmark benchmark to evaluate metadata operations with a relatively larger directory tree than the microbenchmark. Finally, we have collected a trace of Linux kernel compilation and replays file system operations under the limited memory size.

4.2 Resource Consumption

First of all, we examine the amount of memory required for JFFS2, YAFFS2, and DFFS. Figure 2 compares the total memory consumption after each file system is mounted. It is clear that JFFS2 and YAFFS2 show large memory consumption in proportional to the number of total files. Instead, DFFS shows a considerably small memory footprint independent of the number of files. This is because DFFS reads only the minimal information during the mount time, such as super block, checkpoint block, and the root inode.

Figure 3 displays the mount latency. Since JFFS2 and YAFFS2 scan the whole flash memory medium to build the directory structure, it takes up to several seconds to mount a file system. The mount latency is also proportional to the number of files. On the contrary, we can notice that DFFS can be mounted instantly.

4.3 Microbenchmark Results

The microbenchmark initially builds ten randomly-named subdirectories in the root directory where the number of files in each subdirectory is varied from 50 to 200. And then the microbenchmark measures the average time

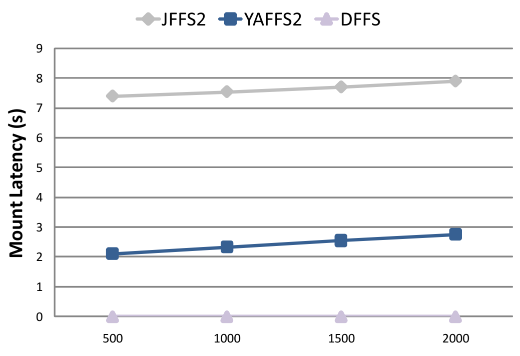

Figure 3. Mount latency by changing the number of total files

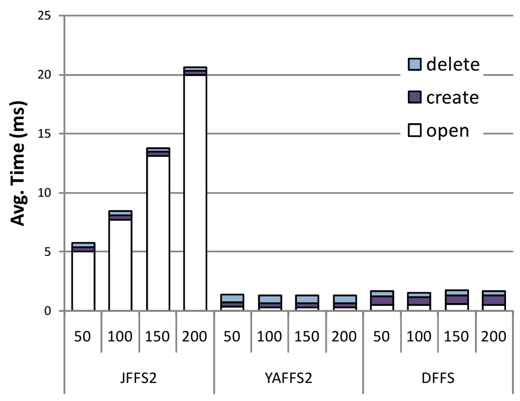

Figure 4. Microbenchmark Results

taken for each open, create, and delete operation. The result is shown in Figure 4.

We observe that the open operation in JFFS2 is significantly slower than in other file systems. This is because, when JFFS2 opens a child inode, all the directory entries of its siblings are also read into memory in order to maximize the locality in directory access. For example, when each directory holds 50 files, opening a file in a directory involves repeated reading of directory entries for other 49 files. This also explains why the open latency increases in proportional to the number of files per directory as shown in Figure 4. It is apparent that JFFS2 performs worse if the file access is randomly distributed over multiple directories.

In contrast, YAFFS2 shows the highest performance since it requires only one flash page read for opening a child inode. DFFS shows the slightly lower performance than YAFFS2 mainly due to hash collisions. However, recall that DFFS consumes substantially less amount of memory (cf. Figure 2).

Compared to the open operation, the create and delete operations have much shorter latencies in all file systems since they require, in most cases, only one or two page writes owing to the log-structured design.

4.4 Postmark Results

Postmark is one of benchmarks that perform intensive metadata operations. It measures the overall throughput in operations/sec (ops/s) while creating and deleting a lot of files in a number of subdirectories. The results are compared in Table 1.

Postmark consists of three phases. In our experiment, Postmark initially builds 100 subdirectories in the root directory. In the first CREATE phase, it creates 1000 files with 128KB of data in a randomly chosen subdirectory. In the second MIXED phase, 2000 create and delete operations are performed randomly. Finally, in the DELETE phase, all the remaining files are deleted.

Table 1. Postmark Results (ops/s)

	JFFS2	YAFFS2	DFFS
Total time (s)	52	48	52
CREATE	38	47	45
MIXED	76	83	68
DELETE	1052	350	350

JFFS2 shows lower create performance than YAFFS2 or DFFS. When a file is created, JFFS2 writes three types of nodes: the child dnode, the parent directory entry, and the parent dnode, while YAFFS2 and DFFS write only the child inode and the parent inode. Although the latency of individual create operation looks roughly same in Figure 4, when summed up, the overall performance impact can be significant for applications that intensively create files.

On the contrary, JFFS2 outperforms YAFFS2 and DFFS by a factor of 3 for the DELETE phase. To delete a file, JFFS2 simply removes relevant data structures in memory and marks the corresponding nodes as invalidated in the spare area. However, YAFFS2 and DFFS require additional time to erase the deleted file's data blocks. In terms of the total elapsed time to run Postmark benchmark, DFFS demonstrates the comparable performance to JFFS2 and YAFFS2.

4.5 Kernel Compilation Trace Results

The kernel trace was collected by hooking system calls which were invoked by metadata operations while compiling the linux kernel. We replayed the trace under 74MB physical memory which was minimum memory size to run YAFFS2. The normalized evaluation results are shown in Figure 5.

Although the open operations in JFFS2 are significantly

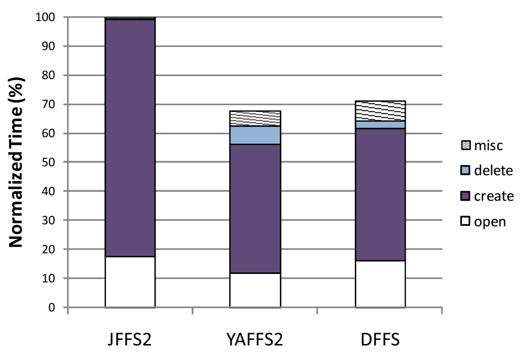

Figure 5. Kernel Compilation Trace Results

slow in the microbenchmark results, they have much better performance in this results. The main reason is from the locality in directory accesses. While the microbenchmark selects a directory randomly to open a file, the kernel trace opens a set of files in the same directory. Therefore JFFS2 can reduce the high open latency which caused by pre-reading directory entries and inodes. Nevertheless it can not show considerably high performance because of reading a lot of directory entries. On the other hand, YAFFS2 shows the highest performance since it maintains the directory tree in the cache even if the system suffers from not enough memory. The DFFS performance also follows the YAFFS2 by 5.1%, since it also does not read a number of directory entries from flash memory by embedding inode. The slight difference, however, is caused by reading some flash pages due to hash collisions in DFFS.

While the create performance is similar to that of Postmark results, the delete one is different, since the deletion in Postmark includes data erase. The delete operations in JFFS2 are optimal by the reasons in Postmark yet, YAFFS2 writes two flash pages composed of deleted inode and updated parent inode. Since DFFS writes only one embedded parent inode page, it shows almost twice better performance than YAFFS2.

The rename performance which is mostly occupied in miscellaneous one, depends on the amount of data to be written. While JFFS2 writes a small directory entry node, YAFFS2 does the renamed inode chunk. On the contrary, DFFS writes both old and new parent directory in addition to the renamed inode.

5 Conclusions

NAND flash memory is one of the most popular storage media for portable embedded systems. As the capacity of NAND flash memory grows, the importance of scalable flash-aware file systems is ever increasing. In this paper, we present the design and implementation of DFFS, a novel flash-aware file system which achieves a small memory footprint and instant mount time. DFFS introduces a new efficient directory structure and an inode indexing scheme. Through comprehensive experimental evaluations, we demonstrate that DFFS has comparable performance to JFFS2 and YAFFS2.

References

[1] A. Bityutskiy. *JFFS3 design issues*. http://www.linux-mtd.infradead.org/tech/JFFS3design.pdf.

[2] Aleph One Company. *Yet another flash file system (YAFFS)*. http://www.aleph1.co.uk/yaffs.

[3] H. Dai, M. Neufeld, and R. Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 176–187, 2004.

[4] F. Douglis, R. Cáceres, M. Kaashoek, K. Li, B. Marsh, and J. Tauber. Storage Alternatives for Mobile Computers. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 25–37, 1994.

[5] E. Gal and S. Toledo. A Transactional Flash File System for Microcontrollers. In *USENIX Annual Technical Conference*, pages 89–104, 2005.

[6] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee. FAB: Flash-Aware Buffer Management Policy for Portable Media Players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, 2006.

[7] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-Memory Based File System. In *USENIX Winter Conference*, pages 155–164, 1995.

[8] J. Kim, J. Kim, S. Noh, S. Min, and Y. Cho. A space efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[9] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *International Conference on Very Large Data Bases (VLDB)*, pages 212–223, 1980.

[10] T. Ohnakado and N. Ajika. Review of device technologies of flash memories. *IEICE Transactions on Electronics*, E84-C(6):724–733, 2001.

[11] D. Roselli, J. Lorch, and T. Anderson. A Comparison of File System Workloads. In *USENIX Annual Technical Conference*, pages 41–54, 2000.

[12] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transations on Computer Systems*, 10(1):26–52, 1992.

[13] Samsung. 256MB SLC NAND Chip Specification. In *K9F2G08R0A @ www.samsung.com*.

[14] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *USENIX Winter Conference*, pages 307–326, 1993.

[15] D. Woodhouse. JFFS: the journalling flash file system. In *Ottawa Linux Symposium*, 2001.

[16] K. Yim, J. Kim, and K. Koh. A fast start-up technique for flash memory based computing systems. In *ACM Symposium on Applied Computing*, pages 843–849, 2005.