Brief Contributions

An Efficient NAND Flash File System for Flash Memory Storage

Seung-Ho Lim, *Student Member*, *IEEE*, and
Kyu-Ho Park, *Member*, *IEEE*

Abstract—In this paper, we present an efficient flash file system for flash memory storage. Flash memory, especially NAND flash memory, has become a major method for data storage. Currently, a block level translation interface is required between an existing file system and flash memory chips due to its physical characteristics. However, the approach of existing file systems on top of the emulating block interface has many restrictions and is, thus, inefficient because existing file systems are designed for disk-based storage systems. The flash file system proposed in this paper is designed for NAND flash memory storage while considering the existing file system characteristics. Our target performance metrics are the system booting time and garbage collection overheads, which are important issues in flash memory. In our experiments, the proposed flash file system outperformed other flash file systems both in booting time and garbage collection overheads.

Index Terms—Flash file system, NAND flash memory, flash translation layer, scan, garbage collection.

1 Introduction

Flash memory has become an increasingly important component in nonvolatile storage media because of its small size, shock resistance, and low power consumption [1]. In nonvolatile memory, NOR flash memory provides a fast random access speed, but it has a high cost and low density compared with NAND flash memory. In contrast to NOR flash memory, NAND flash memory has the advantages of a large storage capacity and relatively high performance for large read/write requests. Recently, the capacity of a NAND flash memory chip became 2GB and this size will increase quickly. Based on the NAND flash chip, a solid state disk has been developed and this can be used as a storage system in laptop computers [2]. Therefore, NAND flash is used widely as data storage in embedded systems and will also be used for PC-based systems in the near future.

NAND flash memory chips are arranged into blocks; each block has a fixed number of pages, which are the units of read/write. A page is further divided into a data region for storing data and a spare region for storing the status of the data region. In first generation NAND flash memory, the typical page size was 512 bytes, the additional spare region was 16 bytes, the block size was 16 KB, and it was composed of 32 pages. As its capacity grew, the page size of the next generation became 2 KB with an additional 64 bytes in the spare region and the block size became 128 KB. Due to flash memory characteristics in the form of Electrically Erasable Read Only Memory (EEPROM), in-place updates are not allowed. This means that, when data is modified, the new data must be written to an available page in another position and this page is then considered a live page. Consequently, the page that contained the old data is considered a dead page. As time passes, a large portion of flash memory is composed of dead pages and the system should reclaim the dead pages for writing operations. The erase operation makes dead pages become available again. However, because the unit of an erase operation is a block, which is much larger than a write unit, this mismatch results in an additional copying of live pages to another location when erasing a block. This process is called garbage collection.

To address these problems, a flash translation layer has been introduced between the existing file system and flash memory [3]. This block level layer redirects the location of the updated data from one page to another and manages the current physical location of each data in the mapping table. The mapping between the logical location and the physical location can be maintained either at the page level (FTL) [4] or the block level (NFTL) [5]. The main differences between these two mapping methods are the mapping table size and redirecting constraints; the FTL has a larger table size, but is more flexible and efficient than the NFTL. The advantage of these methods is that the existing file systems, such as Ext2 and FAT, can be used directly on the flash translation layer. However, the direct use of an existing file system has many performance restrictions and is, thus, inefficient because the existing file systems are designed for disk-based storage systems. For example, the different characteristics between metadata and data, file access patterns, and file size distributions affect the performance of storage systems and these characteristics are crucial for file system designs.

A more efficient use of flash memory as storage would be possible by using a file system designed specifically for use on such devices, without the extra translation layer. One such design is the Journaling Flash File System 2 (JFFS2) [6]. The JFFS2 is a log-structured file system that sequentially stores the nodes containing data and metadata in every free region in the flash chip. However, in the design of the JFFS2, the NAND flash memory characteristics, such as the spare regions and read/write units, were not fully considered and utilized. Therefore, the performance of the JFFS2 with NAND flash memory storage is reduced, especially for the mount time and RAM footprint. The JFFS2 creates a new node containing both the inode and data for the file when the write operation is performed and the corresponding inode's version is increased by one. Therefore, the JFFS2 should scan the entire flash memory media at the mounting time in order to find the inode with the latest version number. Furthermore, many in-memory footprints are required to maintain all the node information.

Another more efficient approach to using flash memory as storage is the Yet Another Flash File System (YAFFS) [7], which is designed specifically for NAND flash memory chips. In YAFFS, each page is marked with a file ID and chunk number. The file ID denotes the file inode number and the chunk number is determined by dividing the file position by the page size. These numbers are stored in the spare region of the NAND flash memory. Therefore, the boot scanning time to build file structures should only require reading of the spare region; thus, the mounting time is faster than that of the JFFS2. However, it also requires full flash scanning to find out the flash usage, so the boot scanning time increases linearly along with the flash memory size. The overall flash memory-based file system architecture is shown in Fig. 1.

These two designs, JFFS2 and YAFFS, are effective for considering the characteristics of flash memory and yield better performance than the flash translation methods because the translation layer between the file system and flash memory chip is not present. However, in designing a flash file system, these two

• *The authors are with the Computer Engineering Research Laboratory, Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, Daejeon Korea.*
E-mail: shlim@core.kaist.ac.kr, kpark@ee.kaist.ac.kr.

Manuscript received 21 Jan. 2005; revised 22 Aug. 2005; accepted 26 Jan. 2006; published online 22 May 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0017-0105.

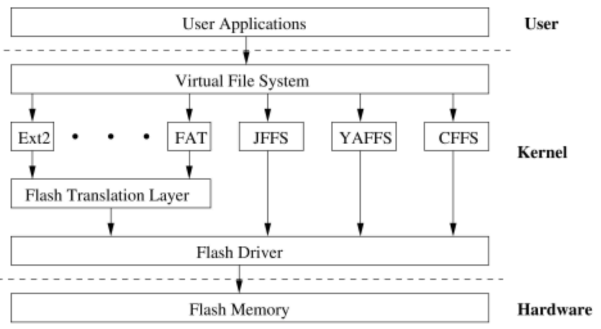

Fig. 1. Flash memory-based file system architecture.

file systems hardly consider the file system characteristics, such as the different characteristics between metadata and data and the file usage patterns according to the file sizes. These characteristics greatly affect the flash memory performance.

In this paper, we present an efficient NAND flash-based file system that addresses both the file system issue and flash memory issue. In the proposed flash file system, we first tried to store the entire data index entries in each inode to reduce the flash scan time. For this, each inode occupied an entire flash page to preserve sufficient indexing space. In addition, we provide two inode classes: direct indexing and indirect indexing. Second, we allocated separate flash blocks for the metadata and data regions which leads to a pseudo-hot-cold separation because metadata are hotter than data. The hot-cold separation can reduce the garbage collection overheads. The remainder of the paper is organized as follows: Section 2 addresses the overall file system architecture and specific design issues. In Section 3, we describe the flash file system analysis in terms of a scan operation and garbage collection. Section 4 presents the performance evaluations and experimental results. Finally, we conclude in Section 5.

2 Flash File System

2.1 File System Design

The design goals of the flash file system, which is called the Core Flash File System (CFFS), are determined by the different access patterns between the metadata and data and the file usage patterns according to the file sizes. The fundamental file system structure of the CFFS has followed that of YAFFS [7]. In the CFFS, each inode occupies an entire page, like YAFFS. Each inode includes its attributes, such as its i-number, uid, gid, ctime, atime, mtime, and so on. In addition to this, the inode stores its file name and parent inode pointer; thus, the CFFS does not have distinct dentries in the media. This can reduce additional flash page updates because a dentry update is not required. For fast lookup in the directory, the CFFS constructs the dentry in the RAM when the system is booted. File data is stored as a chunk whose size is the same as a flash page. Each data chunk is marked with an inode number and a chunk number in the spare region; the chunk number represents the file offset, so it is determined by dividing the file position by the page size. If an inode occupied one page, it would require a high storage capacity compared with other Unix-like file systems, such as Ext2; however, one page per file is not a significant overhead compared to the large data region. Rather, if several inodes share one flash page, as in the Ext2 file system, the update frequency of that page will increase by the number of inodes stored on that page, thus resulting in many flash pages being consumed because some modifications of flash pages require whole page updates. Therefore, the effect of a one page occupation per inode can have a similar effect to the sharing of several inodes on one flash page.

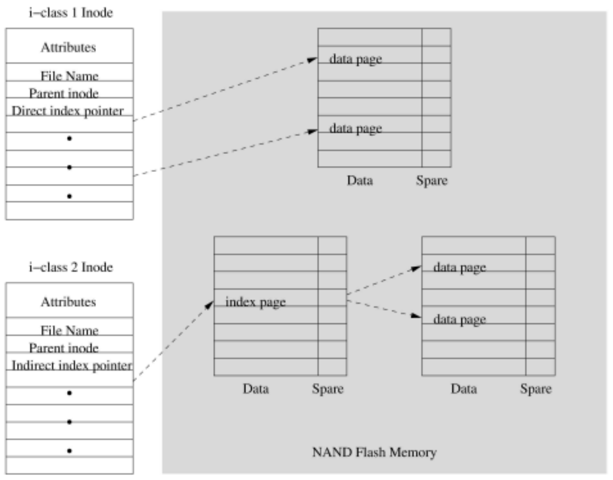

Fig. 2. The different Inode Index Modes between $i-class1$ and $i-class2$ in CFFS.

The main feature of the CFFS is the data index entries in the inode structure. Since an entire flash page is used for one inode, numerous indexing entries can be allocated to point to the data regions. For example, if we use a flash memory with a 512 byte page size, 64 four-byte index entries can exist; if we use a flash memory with a 2 KB page size, 448 four-byte index entries can exist. The four-byte digit number is sufficient to point to an individual flash page. Using these index entries, the CFFS classifies the inode into two classes: $i-class1$ maintains direct indexing for all index entries except the final one and $i-class2$ maintains indirect indexing for all index entries except the final one, as shown in Fig. 2. The final index entry is indirectly indexed for $i-class1$ and double indirectly indexed for $i-class2$. The entry size and related data size range is summarized in Table 1.

The reason the CFFS classifies inodes into two types is the relationship between the file size and usage patterns. Recent studies [14], [16] confirm that most files are small and most write accesses are too small files; however, most storage is consumed by large files that are usually only accessed for reading. In a Unix-like file system, such as Ext2, a fixed inode index strategy is applied to all files and it causes a large portion of index entries in Ext2 to be dedicated to indirect or higher level indexing and only a few index entries are dedicated to direct indexing. For instance, Ext2 allocates only 12 entries for direct indexing, which represents 2.4 percent of the entire space when the file size is 1MB, and the general threshold of the file size between a small file and a large file is 1 MB [16]. Therefore, the probability that the file uses indirect or higher level indexing entries is much higher, even if the file is small. Under those circumstances, writing data will result in additional flash page consumption because every data should be written to a new flash page and it leads to the update of the index information in the inode. Additional page consumption refers to

TABLE 1
Inode Index Mode and File Size Range

Flash Page Size	Metadata Information Size	# of Index Entries	File Size Range	
			i–class 1	i–class 2
512 bytes	256 bytes	64	96KB	12MB
2 KB	256 bytes	448	1916KB	960MB

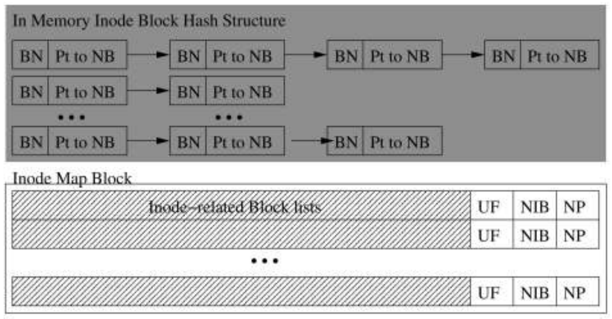
In Memory Inode Block Hash Structure
BN
Pt to NB
Inode Map Block
Inode-related Block lists
UF
NIB
NP

Fig. 3. The $InodeMapBlock$ management.

the pages consumed due to updating the inode. We note that the number of additional flash pages consumed due to updating the inode index information is proportional to the degree of the indexing level. This phenomenon is similar in the case of FAT, in which additional file allocation table and dentry updates are required. When a small file is accessed in the CFFS, the additional flash page consumption is only the inode page itself because most small files are in $i - class1$ and have direct indexing entries. However, each writing of a large file consumes two additional pages because large files are in $i - class2$. For $i - class2$, writing any entry in the indirect indexing entries causes additional page consumption due to updating the indexed page, which contains the real pointers of data. According to recent research results [14], [16], most files are included in $i - class1$ and most write accesses are concentrated in $i - class1$ files in the CFFS. Most operations for large files are read operations and inode updates are rarely performed, so the overheads for indirect indexing in $i - class2$ files are insignificant in the CFFS.

In allocating flash memory blocks, the CFFS writes metadata and data in separate flash blocks with no mixture of the two in a single flash block. The flash blocks are classified into inode-stored blocks, data blocks, and free blocks. By doing this, we can construct the file structures by scanning only the inode-stored blocks when the system boots because the inode has its data index information. The only information required is where the inodes are stored in the flash blocks. To find this, we keep the inode-stored block list in the first flash memory block, called the $InodeMapBlock$. As shown in Fig. 3, the $InodeMapBlock$ consists of an unmount flag (UF), a total number of inode-stored blocks (NIB), the required number of pages to store the inode-stored block list (NP), and the actual inode-stored block list information. The UF, NIB, and NP are stored in the spare region of the $InodeMapBlock$. For integrity, these are repeated through all the pages. The unmount flag indicates that the file system was unmounted cleanly at last unmounting, so the $InodeMapBlock$ contains valid information. The NIB and NP identify how many flash blocks are used for inodes and determine how many pages in the $InodeMapBlock$ should be read. The data region of each page stores the actual inode-stored block list information. We allocated two bytes of space for the block numbers. A two-byte digit number is sufficient to represent the block number. Because a two-byte offset is strictly increased, one page is entirely consumed when (page size)/2 blocks are allocated for the inode. If the entire data region is consumed to store the list information, the next page can be allocated to store the remaining information. The maximum number of blocks that $InodeMapBlock$ can point is (# of pages) x ((page size)/2), which is sufficient to cover all blocks because, currently, most high capacity flash memory contains a maximum of 32,768 blocks. In fact, the inode-stored blocks are much less than the maximum because fewer blocks are used for inodes and most blocks are allocated to store real data. In the main memory, the $InodeBlockHash$ structure is maintained to identify the inode-stored blocks. It is constructed at mounting by reading the $InodeMapBlock$ and continues until the file system is unmounted. The hash key is a simple module number of the $InodeBlockHash$ structure. The management of this is a simple: insertion for a newly allocated block and deletion for a block reclaimed for garbage collection. That is, if a new allocation is required for an inode write, one block is removed from the free block list and inserted into the $InodeBlockHash$ structure or, when garbage collection is performed and the reclaimed block is an inode-stored block, deletion from the $InodeBlockHash$ structure occurs.

At unmounting, the $InodeBlockHash$ should be written to the $InodeMapBlock$ using the UF for a fast scan at the next mount. For that, the $InodeMapBlock$ is erased immediately after the end of mounting. Because the $InodeMapBlock$ is erased once per mounting, it is rarely erased compared with other blocks. Therefore, the $InodeMapBlock$ does not wear out and this ensures the wear-leveling property. However, if the file system is not unmounted cleanly, the $InodeMapBlock$ has useless information and cannot be used. Even if the information is useless, the CFFS mount is faster than others because it can only read the inode-stored blocks by scanning all flash medium.

2.2 Scan Operation at Mounting

While mounting the flash file system, a flash memory scan is required to obtain the flash usage and, accordingly, the file system usage. In the CFFS, scanning involves a two stage operation. First, an $InodeMapBlock$ scan is performed in which the first flash block is checked. If the UF in the $InodeMapBlock$ is set, which means the file system was cleanly unmounted last time, the $InodeMapBlock$ contains valid inode-stored block list information. Otherwise, the file system may have crashed due to an abrupt power failure or system failure at the previous unmounting time, so the $InodeMapBlock$ should not be used in mounting because the last version of the $InodeMapBlock$ is always written immediately before the file system is cleanly unmounted. This checking is repeated while the pages containing the partial inode-stored block list are read. After the $InodeMapBlock$ scan is complete, a second stage is performed. During the second stage, the $BlockInfomation$ and $PageBitmap$ structures are used to maintain information about each block state and each page usage within a block, respectively. If the $InodeMapBlock$ is valid and the $InodeBlockHash$ is constructed during the first stage, the inode-stored blocks are scanned through the $InodeBlockHash$. For each page within an inode-stored block, we identify whether the page contains valid inode or not by reading the spare region. If it contains valid inode, we read the page with the CRC check and allocate an inode cache structure in the main memory to construct a directory relation. The allocated inode cache structures are connected to each other to make directory structures using their parent, sibling, and children pointers. After checking the attributes, the data pages are checked. For the $i - class1$ inodes, we check the data pages directly using direct index entries and mark them to the $BlockInfomation$ and $PageBitmap$ data structures. For the $i - class2$ inodes, we check the data pages with the traverse indexed pages through the indirect index entries, which requires more page scans. Each indexed page is checked and the $BlockInfomation$ and $PageBitmap$ are updated accordingly.

If the $InodeMapBlock$ is not valid, which means the file system was not cleanly unmounted the last time, we must check every block to find whether it stores inodes or not to find the inode-stored blocks. If the block being checked is an inode-stored block, the same operation as described above is applied. If it is not, the block does not need to be scanned. Because block checking is performed only by the reading spare region of the first page for each block, the scanning operation is much faster than that of other flash file systems.

2.3 Operation

During the scan operation, we are able to categorize the blocks into three types: inode-stored blocks, data blocks, and free blocks. In addition, we can check which pages have valid inodes in the inode-stored blocks and which pages have valid data in the data blocks. This is sufficient to make a normal file operation because the CFFS is based on the principle of a journaling mechanism for each block write. This means that if some blocks are not checked in the $BlockInfomation$ structure, they are not allocated to be either an inode-stored block or a data block: Thus, these blocks are considered free blocks. Similarly, if one or more pages are not checked in the $PageBitmap$, they are considered invalid pages, i.e., dead pages. The remaining task is to make the starting block pointer and page pointer for the first write operation after the system boots.

For each file, because the inode has information on the location of its own position and its data positions in the flash memory, the CFFS does not need to maintain all data pointers for every file in memory, unlike other flash file systems. This can reduce memory footprints; however, additional inode page reads should precede the data read/write operations, which affect the user latency. To compensate for this, the CFFS manages a cache list for files that are frequently accessed. The files which are in the cache maintain their data index information in the main memory. By doing this, the CFFS can locate the data region without additional page readings for inode if the file is in the cache list. The cache size and management is a performance configuration parameter for the trade-off between the memory footprint and read/write latency.

File operation according to the inode classification is as follows: When a file is created, the file is first set to $i-class1$ and it is maintained until all index entries are allocated for the file data. As the file size grows, the inode class is altered from $i-class1$ to $i-class2$ when there is no indexing pointer in $i-class1$. This alteration can be performed naturally during runtime with little cost because every inode update should be written to another clean region in the flash memory. The only thing that remains to be done is the reorganization of the indexing entries from direct to indirect indexing.

2.4 Garbage Collection

Unlike the disk-based file system, a flash memory-based file system is free from seek latency and is expected to show comparable performance for both sequential and random read operations. The write performance, on the other hand, is affected by additional page consumption and garbage collection. A flash memory storage system cannot be efficient without the support of a good garbage collection operation. Garbage collection should occur when a block contains only dead pages or there are insufficient free blocks available. During this process, any live pages in the reclaimed block must be copied to an available space.

Previous research has focused on the hot-cold aware garbage collection policies [9], [13]. Hot data represents data that has a higher probability for updates in the near future than cold data. Pages that store hot data usually have a higher chance of becoming dead pages in the near future. From the file system's viewpoint, it is a well-known fact that metadata, i.e., inodes, are much hotter than regular data. Anytime and anywhere, a write operation of a file results in its inode being updated to maintain file system integrity. Even when there is no writing of data, the inode sometimes should be modified for several reasons: file movement, renaming operations, file attribute modifications, and so on. In a sense, the CFFS already uses a pseudo-hot-cold separation by allocating different flash blocks for metadata and data with no mixture between these two in a single block and with low maintenance overheads. Therefore, we can perform efficient garbage collection by separating the metadata and data in different flash blocks. The relatively "hot" inode pages are stored in the same block so that the amount of copying of hot-live pages can be minimized. Garbage collection is also different according to the type of reclaimed block. If a reclaimed block is an inode-stored block, the live inode pages in the block will be copied to other available inode-stored blocks; if the reclaimed block is a data block, the live data pages will be copied to other available data blocks.

According to our proposed flash block separation and garbage collection policy, selecting an inode-stored block for reclamation seems to be more frequent; thus, there seems to be too much erasing of inode-stored blocks compared with data blocks and this may lead to a wear-leveling problem. To avoid this problem, we set a weight value for each block when the block is selected to be erased. If the block is allocated as an inode-stored block one time, it will be allocated as a data block the next time by using the weight value.

3 System Analysis

In this section, we present the results obtained from our flash file system analysis. First, we describe the scan operation. We assume that the file system has a total of n files and each file occupies S pages on average. For each page, the read time of the data region and spare region is denoted by T_d and T_s, respectively. Also, let p denote the page size in bytes. From the above, the total allocation of the file system in flash memory is approximately nS pages. For a comparison, we analyze the scan time of YAFFS. Let us denote the scan time of YAFFS by T_y. YAFFS requires scanning of both the metadata and data pages upon mounting. During scanning, the inode pages are read in both the data region and spare region of the page, while the data pages are read only in the spare region of the page. Therefore, the time taken for nS pages to be scanned can be expressed as:

$$T_y = n(T_d + T_s) + nST_s. \tag{1}$$

In the CFFS, file size distribution is an important parameter because the inode is classified by its file size. For simplicity, we assume that $i-class1$ files occupy S_1 flash pages on average and the portion of $i-class1$ files is δ. Also, $i-class2$ files occupy S_2 flash pages on average and the portion of $i-class2$ files is 1-δ. For $i-class1$ files, only the inode page scans are required. For $i-class2$ files, each indirect indexed page should be scanned because they have the real data page pointers. The amount of data allocation per indirect indexed page is $p/4$ when each page is indexed by a 4-byte pointer. The CFFS only scans the inode-stored blocks, so the scan time of the CFFS, T_c, can be expressed as:

$$T_c = n\left(\delta + (1-\delta)\frac{4S_2}{p}\right)(T_d + T_s), \tag{2}$$

where

$$S = \delta S_1 + (1-\delta)S_2. \tag{3}$$

In (2), the scan time is dependent on the file number and fraction of $i-class2$ files. Since most files are small and dedicated to $i-class1$, δ is close to 1. In general, δ is about 0.8 or above. Therefore, the second term of (2) will be small even though S_2 implies many flash page reads. It is obvious that the time T_c is shorter than T_y in most cases.

Next, we describe garbage collection. In the CFFS, we denote the inode block by B_I and the data block by B_D. Accordingly, let P_I denote the inode page and P_D denote the data page. The probability of each data page being invalid for a file can be denoted by $P(P_{D,k})$, where $1 \leq k \leq S$. Since every data write leads

TABLE 2
Flash Memory System Environment for Experiments

Experimental Environment	NAND Flash Memory Characteristics
Platform : SMDK2410	Part Number : K9S1208VOM
CPU : ARM9 200MHz	Block Size : 16KBytes (4096 blocks)
Memory : SDRAM 64MB	Page Size : (512 + 16)Bytes
Flash Memory : NAND	Page read time : $12\mu s$
OS : Linux 2.4.18	Page write time : $200\mu s$
MTD : nand.c	Block Erase time : 2ms

to an update of the inode, we can assume that the probability of each inode page being invalid, $P(P_I)$, is higher than the $P(P_D)$ for each write operation. In each inode-stored block, let us assume that each page has the same probability for updates. Although the assumption that each page has the same probability is unreal, it is meaningful that the inode-stored block has a much higher probability to be reclaimed than the data block for garbage collection. On the other hand, if the reclaimed block has valid pages, the pages should be copied to an available flash memory region. Therefore, the overhead of the garbage collection is the number of valid pages in the reclaimed block and is conversely proportional to the reclamation probability.

4 Performance Evaluation

In this section, we evaluate the performance of the CFFS flash file system. We compared the CFFS with JFFS2, YAFFS, and other flash memory-based storage systems, including FTL and NFTL. Ext2 and FAT are built on the FTL and NFTL block level device drivers because they cannot operate by themselves. We implemented the CFFS in a Linux OS with kernel version 2.4.18. The experimental platform consists of an embedded Linux system platform with a 200 MHz ARM processor and 64 MB of main memory. The NAND flash memory used in this experiment is K9S1208VOM [2] SMC NAND flash memory. The NAND flash memory system environment for experiments is described in Table 2.

At first, we compared the scan time. In the scanning operation, flash file systems are required to obtain the file system level information, so scan time varies with the file system usage. We filled the file system with various data whose distribution was from several bytes to several MB. Many files were copied to a flash medium to fill the storage. The distribution was followed by the general computing environment. Most files were small and, thus, required many metadata and most storage was consumed by a few large files. Table 3 describes the scan time for each flash-based storage system. The first two systems are FTL and NFTL; because these two systems are not required to obtain file system level information, the scan time is fixed with a flash memory size. For 64 MB flash memory, the scan times are estimated to be 1.06 s and 808 ms for FTL and NFTL, respectively. The later three systems are concerned with the flash file systems. To maintain equality between all systems, we estimated the scan time for a file system usage with 64 MB. As shown in Table 3, the CFFS showed much faster scan operations than other systems. This is because the CFFS only scans the inode-stored blocks, the small region of the flash medium, as described in Section 2.

Next, we evaluated the garbage collection (GC) performance. As explained in Section 2, the read operation does not cause needless flash page consumption. Therefore, the read is not the target performance metric. Write performance is affected by garbage collection; therefore, the proper benchmark for evaluating GC should have many write and create/delete transactions, which will cause many validations and invalidations of pages and lead to a GC operation. We used the PostMark [15] benchmark program, which is a well-known benchmark for evaluating file systems and is a proper benchmark for evaluating GC. PostMark creates a set of files with random sizes within a range set. The files are then subjected to transactions consisting of pairing of file creations or deletions with file reading or writing. Each pair of transactions is chosen randomly and can be biased via the parameter settings. We ran our experiments using the configuration of 100-500 files with a file size range of 512 bytes to 60 KB. One run of this configuration performs approximately 1,000 transactions with equal probability of create/delete and read/write operations. The ratio between number of files and the number of transactions varies between 1:10, 1:3, and 1:2 for every run and the results are collected according to the ratio. Since the purpose of our experiments is not the comparison of the caching of the buffer cache or page cache, we avoided the cache effect by minimizing the $bdfush$ time.

TABLE 3
Experimental Results for Scan Operation: Comparision between FTL, NFTL, JFFS2, YAFFS, and CFFS

FFS Type	FTL	NFTL	JFFS2	YAFFS	CFFS
Scan Time	$1.06s$ $(Fixed)$	$808ms$ $(Fixed)$	$22.6s$	$3s$	$593ms$

Fig. 4 shows the write frequency distribution for each file system test. For all figures, the x-axis shows the logical block address with a range of 0 to 2,000 for each file system and the y-axis represents the write counts with different ranges. One logical block address matches one flash page and the logical address means the file system viewpoint. From the figures, Ext2 and FAT show a high update frequency in the small narrow region; this hot region is related to the metadata. As the file transaction ratio becomes higher, the update frequency is higher and the region becomes smaller. However, YAFFS and the CFFS show that the update frequency is spread throughout the entire region of the logical block address. The reason behind Ext2 and FAT having higher update counts in the small region than YAFFS and the CFFS is that several inodes or dentry lists in Ext2 or FAT share the same flash page, that is, the density of page sharing of inodes.

From the above, we identify that, from the perspective of NAND flash memory, the small inode size has little meaning because the updates of other inodes in a shared page will result in the invalidation of that page. In the cases of CFFS and YAFFS, many update of metadata in the CFFS and YAFFS still exist in spite of the spreading metadata update region. This is the nature of metadata and data as we analyzed in Section 3. Therefore, our metadata and data separation is still worthwhile for flash block allocation. Table 4 describes the number of extra page writes per garbage collection. As shown in Table 4, the flash file system approaches outperform the existing general file system approaches because the flash file system has the garbage collection algorithm based on flash usage while considering the file system information. In addition, the inode-data block separation method gives better performance than YAFFS, by approximately 9 percent to 20 percent, in garbage collection overheads.

Finally, we summarized the performance comparison of the JFFS2, YAFFS, and CFFS; Table 5 presents the summary. The results are from the experiments using PostMark with a transaction ratio 1:2. The first item shows that the CFFS has a much faster scan time than other systems. Second, the PostMark transaction rate indicates the number of transactions per second performed by PostMark, which implies the read/write throughput. In the results, YAFFS and the CFFS give a slightly better performance than JFFS2. This is because JFFS2 writes nodes containing both the inode and data which cause some overheads in inode buildup and file storing. Also, JFFS2 writes data by compression. The JFFS2 complex operation for storing/retrieving data to/from flash memory gives poor I/O performance. In the cases of YAFFS and CFFS, the

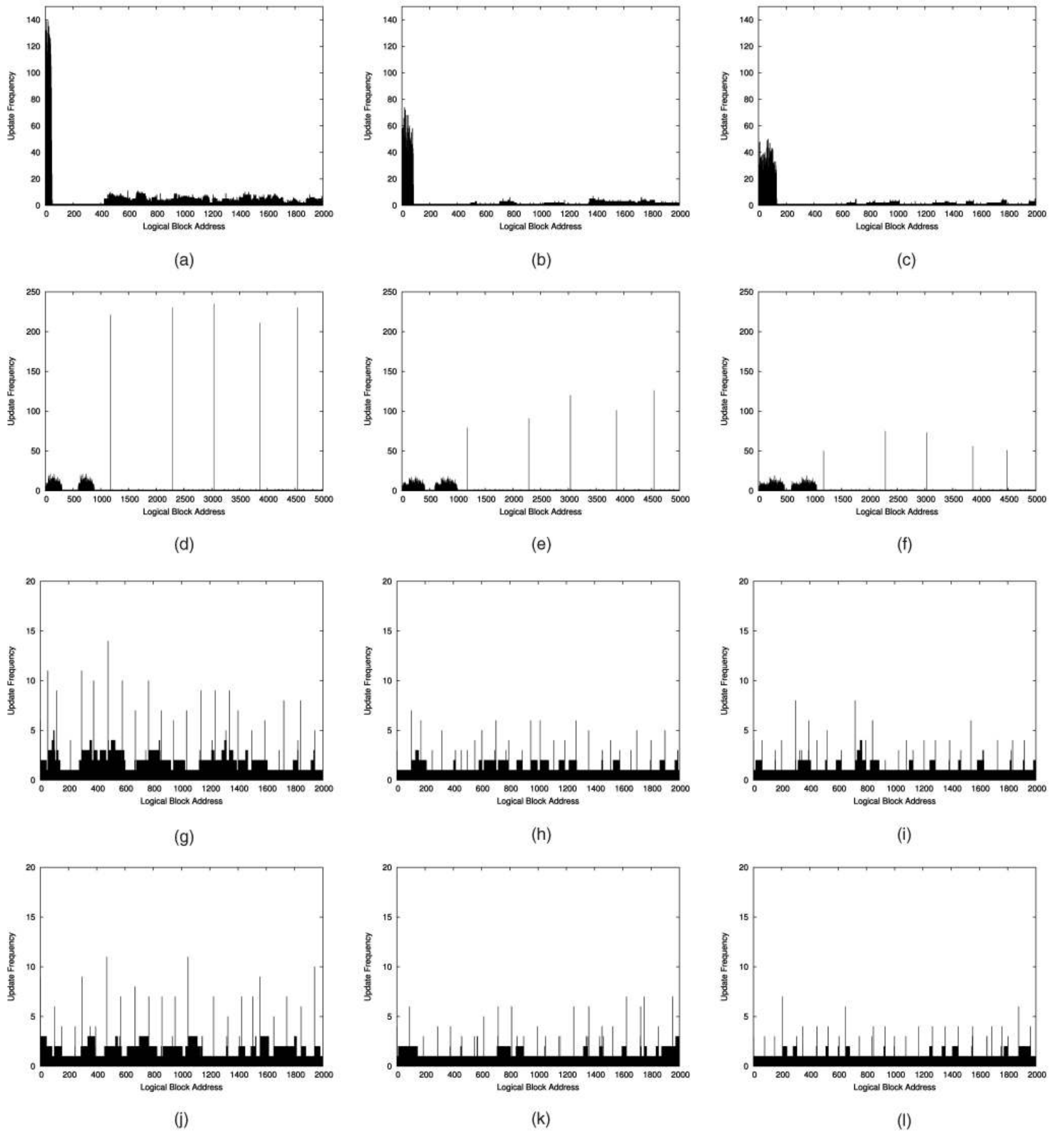

Fig. 4. Write access patterns for postmark benchmark; file transaction ratio with 1:10, 1:3, 1:2, respectively. For each ratio, Ext2 over FTL, FAT over FTL, YAFFS, and CFFS perform postmark benchmark. (a) Ext2 over FTL for 1:10. (b) Ext2 over FTL for 1:3. (c) Ext2 over FTL for 1:2. (d) FAT over FTL for 1:10. (e) FAT over FTL for 1:3. (f) FAT over FTL for 1:2. (g) YAFFS for 1:10. (h) YAFFS for 1:3. (i) YAFFS for 1:2. (j) CFFS for 1:10. (k) CFFS for 1:3. (l) CFFS for 1:2.

TABLE 4
Performance Comparison of Garbage Collection for NFTL, FTL, YAFFS and CFFS

File Transaction Ratio	# of Extra Writes per GC			
	Ext2 over NFTL	Ext2 over FTL	YAFFS	CFFS
1 : 10	3.69	0.70	0.30	0.26
1 : 3	3.37	0.69	0.17	0.15
1 : 2	2.47	0.71	0.15	0.12

TABLE 5
Comparison of Flash File Systems: JFFS2, YAFFS, and CFFS

Parameters	JFFS2	YAFFS	CFFS
Scan Time	$22.6s$	$3s$	$593ms$
Postmark Transaction Rate	0.426	0.485	0.487
# of Flash Page Writes	90045	68055	68689
# of Extra Writes per GC	0.471	0.148	0.119

postmark transaction rate is slightly better than JFFS2. However, the I/O performance should be an improved performance factor in comparison with an FTL-based system. The third represents the total number of written pages while performing the benchmark. When comparing between CFFS and YAFFS, CFFS has a larger write operation than YAFFS, which is related to the writing of indirect indexed pages for large files. This is the weak point of CFFS and this overhead increases as the portion of large files is increased. Finally, the extra write operation per GC is reduced by about 9 percent to 20 percent in garbage collection overhead of YAFFS.

5 Conclusion

In this paper, we presented a flash file system designed for NAND flash memory storage. Unlike the disk-based file system, flash memory-based storage is free from seek latency and is expected to show comparable performance for both sequential and random read operations. The write performance of flash memory is affected by garbage collection because all written data should be redirected to a clean page and this leads to the erase operations for dirty blocks. The proposed flash file system, called the CFFS, uses a pseudo-hot-cold separation by allocating the different flash blocks for metadata and data. The metadata and data separation method improves garbage collection performance more than in other methods. In addition, we classified the inodes using indexing entries: $i-class1$ for direct indexing entries and $i-class2$ for indirect indexing entries. The CFFS allocates one page per inode, which yields numerous indexing entries to allocate all the data page pointers for small files, although large files are required to use the indirect indexing method with $i-class2$. This inode indexing method and metadata-data block separation results in a fast scan time when the file system is mounted and garbage collection is efficient.

Acknowledgments

The authors would like to thank Joo-Young Hwang for support of the design and implementation of the scheme. This work was supported by Samsung Electronics.

References

[1] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J.A. Tauber, "Storage Alternatives for Mobile Computers," *Proc. First Symp. Operating Systems Design and Implementation (OSDI),* pp. 25-37, 1994.

[2] Sansung Electronics Co., "NAND Flash Memory & SmartMedia Data Book," 2002, http://www.samsung.com/.

[3] "Memory Technology Device (MTD) Subsystem for Linux," http://www.linux-mtd.infradead.org, 2004.

[4] Intel Corp., "Understanding the Flash Translation Layer (FTL) Specification," http://developer.intel.com/, 1998.

[5] A. Ban, "Flash File System," US Patent, no. 5,404,485, Apr. 1995.

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

[7] Aleph One Ltd, Embedded Debian, "Yaffs: A NAND-Flash Filesystem," http://www.aleph1.co.uk/yaffs/, 2002.

[8] R. Card, T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," *The HyperNews Linux KHG Discussion,* http://www.linuxdoc.org, 1999.

[9] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," *Proc. Usenix Technical Conf.,* 1995.

[10] M. Rosenblum and J.K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Computer Systems,* vol. 10, no. 1, 1992.

[11] D. Kabushiki, "Flash Memory Card with Block Memory Address Arrangement," US Patent no. 5,905,993, 2001.

[12] L.P. Chang and T.-W. Kuo, "An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems," *Proc. ACM Symp. Applied Computing,* Mar. 2004.

[13] L.P. Chang and T.W. Kuo, "A Real-Time Garbage Collection Mechanism for Flash Memory Storage System in Embedded Systems," *Proc. Eighth Int'l Conf. Real-Time Computing Systems and Applications,* 2002.

[14] D. Roselli, J.R. Lorch, and T.E. Anderson, "A Comparison of File System Workloads," *Proc. 2000 USENIX Ann. Technical Conf.,* June 2000.

[15] J. Katcher, "PostMark: A New File System Benchmark," Technical Report TR3022, Network Applicance Inc., Oct. 1997.

[16] A.-I.A. Wang et al., "Conquest: Better Performance through a Disk/Persistent-RAM Hybrid File System," *Proc. 2002 USENIX Ann. Technical Conf.,* June 2002.