ScaleFFS: A Scalable Log-Structured Flash File System for Mobile Multimedia Systems

DAWOON JUNG, JAEGEUK KIM, JIN-SOO KIM, and JOONWON LEE
Korea Advanced Institute of Science and Technology

NAND flash memory has become one of the most popular storage media for mobile multimedia systems. A key issue in designing storage systems for mobile multimedia systems is handling large-capacity storage media and numerous large files with limited resources such as memory. However, existing flash file systems, including JFFS2 and YAFFS in particular, exhibit many limitations in addressing the storage capacity of mobile multimedia systems.

In this article, we design and implement a scalable flash file system, called ScaleFFS, for mobile multimedia systems. ScaleFFS is designed to require only a small fixed amount of memory space and to provide fast mount time, even if the file system size grows to more than tens of gigabytes. The measurement results show that ScaleFFS can be instantly mounted regardless of the file system size, while achieving the same write bandwidth and up to 22% higher read bandwidth compared to JFFS2.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Memory technologies*

General Terms: Design, Performance

Additional Key Words and Phrases: File system, flash memory, NAND, storage system

ACM Reference Format:

Jung, D., Kim, J., Kim, J.-S., and Lee, J. 2008. ScaleFFS: A scalable log-structured flash file system for mobile multimedia systems. ACM Trans. Multimedia Comput. Commun. Appl. 5, 1, Article 9 (October 2008), 18 pages. DOI = 10.1145/1404880.1404889 http://doi.acm.org/10.1145/ 1404880.1404889

1. INTRODUCTION

During the past few years, there has been explosive growth in the sales of portable multimedia systems such as MP3 players and portable media players [Paulson 2005]. As their functions and capabilities become more diverse and complex, they demand increasingly large storage capacity. Mobile multimedia systems such as MP3 players, portable media players (PMPs), and digital camcorders require huge storage space to store still pictures, audio, and video files. Unlike other general purpose systems, these multimedia systems usually deal with large files. For example, the size of a 4-minute-long MP3 audio

This research was supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Advancement) IITA-2008-C1090-0801-0020.

Author's address: D. Jung, J. Kim, J.-S. Kim, J. Lee, Computer Science, Department, KAIST, 335 Gwahangno, Yuseong-gu, Daejeon 305-701, Republic of Korea; email: {dwjung, jgkim}@calab.kaist.ac.kr; J.-S. Kim, J. Lee, School of Information and Communication Engineering, Sung Kyunkwan University, 300 Cheoncheon-dong, Jangan-gu, Suwon, Gyeonggi-do 440-746, Republic of Korea; email: {jinsoo kim, joowon}@skku.edu.

file encoded in 128Kbps is 3.8MB and the size of 1Mbps one-hour video is 450MB. Typically, 1–10 MB of audio and image files and more than 100MB video files are exchanged in a P2P file sharing system [Gummadi et al. 2003; Saroiu et al. 2002]. Saroiu et al. [2002] also showed that the typical size of audio files shared in Napster is 3.7MB on average and files from several kilobytes to hundreds of megabytes are shared. In order to handle these large files, the systems need to manage large storage media space as well as numerous large files. In addition, mobile multimedia systems should operate with limited resources.

Flash memory has become an attractive storage medium for mobile multimedia systems due to its outstanding characteristics such as nonvolatility, small and lightweight form factor, solid-state reliability, and low power consumption [Douglis et al. 1994; Marsh et al. 1994; Kim et al. 2002]. Among various flash memory types, NAND flash memory provides higher density with low cost, and has hence become standard data storage for portable devices. Recently, the capacity of NAND flash memory has been dramatically increased. A 32Gbit NAND flash chip is already commercially available, and portable devices equipped with several gigabytes of NAND flash memory are widely available. The growth in NAND flash memory capacity is expected to continue or even accelerate in the coming years.

In spite of offering many advantageous features, flash memory has unique operational characteristics. Most notably, in-place updates are not possible in flash memory, that is, previous data should be erased first in order to overwrite other data in the same location. An erase operation is performed in a unit of *flash erase block*, which is much larger than a read and write unit, and the operation has much higher latency than read or write operations. Obviously, it is unacceptable to erase a whole erase block whenever a portion of the block is updated. Another limitation of flash memory is that the number of erase/write cycles allowed for a single erase block is limited to 10,000–1,000,000 times. Thus, it is necessary to perform *wear-leveling*, a scheme to distribute erase/write cycles evenly across all erase blocks in order to extend the lifetime of flash memory storage.

As NAND flash memory shows radically different characteristics compared to hard disks, many studies focusing on using NAND flash memory efficiently as part of the storage system have been conducted [Marsh et al. 1994; Kim et al. 2002; Woodhouse 2001; AlephOne Ltd. 2003]. In particular, building an efficient file system over NAND flash memory presents a considerable challenge, since legacy disk-based file systems cannot be directly used for NAND flash memory. Recently, several small-scale flash-aware file systems, such as JFFS2 [Woodhouse 2001] and YAFFS [AlephOne Ltd. 2003], have been proposed to cope with the unique characteristics of NAND flash memory. These flash-aware file systems work directly on NAND flash memory, performing block erase and wear-leveling as needed.

The present work is primarily motivated by the fact that the architecture of traditional flash-aware file systems, especially JFFS2 and YAFFS, is not appropriate for large-capacity flash storage. When using JFFS2 or YAFFS for mobile multimedia systems with large-capacity NAND flash memory, the following problems are confronted. First, they consume too much memory by keeping all index structures in memory to find the latest file data on flash memory. Obviously, the larger memory requires more electrical power [Huang et al. 2003; Park et al. 2004]. Second, they use too much time to mount a file system, since the entire flash media are scanned to construct the index structures at the mount time. This could lead to frustration for mobile multimedia system users, since they face a considerable wait until their devices become ready.

In JFFS2 and YAFFS, the mount time and the memory consumption are linear functions of the file system size and the amount of the stored data. This stems partly from these file systems having been originally designed for small flash memory, where the overhead can be negligible. However, the slow mount time and the large memory footprint are becoming increasingly serious problems as the size of flash memory scales upward into the gigabyte range.

Another popular approach used for flash memory-based storage is employing a software layer called FTL (Flash Translation Layer) [Marsh et al. 1994; Kim et al. 2002; Kang et al. 2006] that emulates common block I/O interfaces. Owing to this layer, most systems can use legacy disk-based file systems to flash memory-based storages without any kernel or file system modifications. While this layered approach makes it easy to deploy flash memory-based storage devices, FTL can make bad decisions when managing flash memory, because liveness information is not available within storage systems [Sivathanu and Bairavasundaram 2004]. As a result, the performance of flash memory-based storage using FTL is degraded [Lim and Park 2006].

In this paper, we present ScaleFFS, a scalable flash file system for large-capacity NAND flash memory, to address the aforementioned problems of conventional flash-aware file systems. ScaleFFS is designed to support flash memory capacity of more than tens of gigabytes, while requiring only a small fixed amount of memory space. ScaleFFS does not keep its index structures permanently in memory and does not scan the entire flash media to build index structures or directory trees. In addition, ScaleFFS fully utilizes the kernel's cache mechanism for fast file system operation and flexible memory management.

The remainder of this paper is organized as follows. We provide background and discuss related work in Section 2. In Section 3, we present the design and implementation of ScaleFFS. We describe detailed file system operations of ScaleFFS in Section 4. Section 5 presents our experimental results, and we conclude in Section 6.

2. BACKGROUND AND RELATED WORK

2.1 The Organization of NAND Flash Memory

A NAND flash memory chip consists of a fixed number of erase blocks. The first generation of NAND flash memory has 32 pages per erase block. Each page comprises 512bytes of data area and 16bytes of spare area. The spare area is usually used to store management information and error correcting code (ECC). As flash memory technology has improved, a new generation of NAND flash memory, called large block NAND, has been introduced to provide higher capacity. In large block NAND, an erase block consists of 64 pages. The page size is also increased by 4 times; each page contains 2KB of data area plus 64bytes of spare area. Note that, in NAND flash memory, read and write operations are performed on a page basis, while an erase operation is done on a much larger erase block basis.

2.2 JFFS2 (Journaling Flash File System, Version 2)

JFFS2 is a log-structured file system designed for use with flash memory in embedded systems [Woodhouse 2001]. JFFS2 was originally designed and optimized for small NOR flash, and the support for NAND flash was added later.

JFFS2 uses a variable-sized unit called a *node* to store file system metadata and data. The maximum node size is restricted to half the size of a flash erase block, and each flash block can contain one or more nodes. In JFFS2, there are three types of nodes, and each node has its own header. First, a `JFFS2_NOTETYPE_INODE` node contains all the inode metadata, as well as a range of file data belonging to the inode. File data may be compressed using a compression algorithm that can be plugged into the JFFS2 code. Currently, JFFS2 supports three compression algorithms, ZLIB, RTIME, and RUBIN. Typically, a single file consists of a number of `JFFS2_NODETYPE_INODE` nodes. Another node type is `JFFS2_NODETYPE_DIRENT`, which is used to represent a directory entry. A `JFFS2_NODETYPE_DIRENT` node contains the inode number of the directory, the name of the directory or the file, the inode number of its parent directory, etc. This information is used to construct the directory tree of JFFS2. The final node type is `JFFS2_NODETYPE_CLEANMARKER`. When an erase block is successfully erased, this node is written to that block so that JFFS2 can safely use the block as free space.

JFFS2 maintains several lists to manage the status of each erase block: `clean_list`, `dirty_list`, and `free_list`. `Clean_list` contains only the blocks that are full of valid nodes, and blocks on the `dirty_list` have at least one obsolete node. If JFFS2 erases a block, the block is added to the `free_list`.

Since JFFS2 does not have a fixed index structure to locate nodes in flash memory, JFFS2 keeps the pertinent and minimal information in memory for each node. JFFS2 maintains a data structure that consists of the physical address on flash memory, the length of a node, and pointers to the next structure related to the same file so as to be able to retrieve further information later. JFFS2 also retains some hints in memory to traverse the full directory tree, because a directory entry does not have any pointers to child entries.

These in-memory data structures consume considerable memory space, especially when the file system size grows. The memory footprint of JFFS2 is usually proportional to the number of nodes, and is also increased as the size of the file system contents becomes larger. For instance, JFFS2 requires more than 4MB of memory space to construct these data structures for 128MB data if each node size is 512bytes.

On the other hand, these in-memory data structures cannot be built cost-free. They are constructed at mount time by scanning the entire flash memory media. This procedure takes from several seconds to tens of seconds, depending on the number of nodes in the file system. In addition, there exists runtime overhead, because the full information on a file or a directory such as the offset within the file, the latest file attributes, and the file name list of the directory is retrieved by reading every node header from flash memory that belongs to the file or the directory upon the first access to them. This further exacerbates the problem.

2.3 YAFFS (Yet Another Flash File System)

YAFFS is another variant of a log-structured file system designed for NAND flash memory [AlephOne Ltd. 2003]. The structure of YAFFS is similar to that of the original version of JFFS. The main difference is that node header information is moved to NAND's spare area and the size of a data unit, called a *chunk*, is the same as the page size in NAND flash memory to efficiently utilize NAND flash memory.

Similar to JFFS2, YAFFS keeps data structures in memory for each chunk to identify the location of the chunk on flash memory. YAFFS also maintains the full directory structures in memory, since a chunk representing a directory entry has no information about its children. In order to build these in-memory data structures, YAFFS scans all spare areas across the whole NAND flash memory. Therefore, YAFFS inherits the same problems as JFFS2. Moreover, YAFFS is not scalable, because various internal data structures are compacted to support only up to 1GB of flash memory, 512MB per file, and total 256,000 files. In practice, YAFFS can directly cover only 32MB of flash memory due to its 16-bit index structure and it requires an additional lookup operation be used with larger size flash memory.

2.4 LFS (Log-Structured File System)

As presented in the previous subsections, the basic structures of JFFS2 and YAFFS are based on a log-structured file system (LFS). LFS was originally proposed as a disk-based file system in order to efficiently utilize a disk's raw write bandwidth and to provide fast crash recovery [Rosenblum and Ousterhout 1992; Seltzer et al. 1993]. When LFS was first introduced, other Unix file systems showed poor write performance due to disk seek operations, and they relied on `fsck`, which needed to scan the entire disk to recover file system consistency.

The basic concept of LFS is to improve the file system performance by storing all file system data in a sequential structure called a *log*. LFS buffers all writing operations in memory and flushes them to a log sequentially at once if its buffers are full or a sync operation is requested by a kernel. Figure 1 shows how data and index structures are stored in LFS. LFS increases the write performance by eliminating

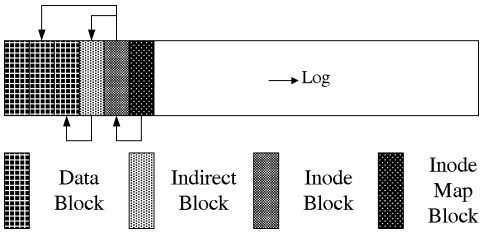

Fig. 1. Log structure of LFS.

almost all disk seeks. The log structure also allows crash recovery that is as fast as that of other logging schemes.

Data structures used in LFS, such as *inode*, *indirect block*, and *directory block*, are similar to those in the Berkeley Unix fast file system [McKusick et al. 1984] or Linux ext2/ext3 file system [Card et al. 1994]. A data block stores file contents, and a directory block has a list of subdirectories and/or file names with their corresponding inode numbers. An inode has file metadata such as file permission, owner, size, etc., and points to data blocks associated with a file. Indirect blocks are used to locate all data blocks for large files. For a given file, every data block can be located by the inode and, if necessary, by indirect blocks. Since an inode block itself is floating in the log, as shown in Figure 1, LFS uses a data structure called an *inode map* to maintain the current locations of all inodes. The inode map occupies 0.2% of live data and is usually cached in memory during file system operations.

One of the most important design issues in LFS is how to maintain large free extents for writing new logs. LFS divides the disk into large fixed-extents called *segments*. The log is written to a segment sequentially from the segment's beginning to its end. Once the log fills the current segment, the log continues at another segment. In LFS, segments are logically threaded, that is, any segment may be chosen to be the next segment provided that it contains no live data. The segment size is either 512KB or 1MB.

LFS also performs segment cleaning to generate free segments. In a segment, there could be live data, as well as obsolete data caused by file deletion or modification. During segment cleaning, LFS first reads one or more segments to be reclaimed and identifies live data by examining segment summary blocks. A segment summary block is a data block containing a file inode number and offset for each block stored in a segment. LFS then copies live data to other free segments and mark old segments as clean. This cleaning procedure is repeated until a number of clean segments (typically 50–100 segments) are produced. For better cleaning efficiency, LFS summarizes segment usage information in the segment usage table and refers to it to choose victim segments.

In order to make file system start and recovery faster, LFS stores the information necessary for the file system mount in a fixed, known location called the *checkpoint region*. During the checkpoint operation, LFS flushes all modified data to the log, and writes the locations of all the inode map blocks, the segment usage table blocks, and a pointer to the last written segment into the checkpoint region. The checkpoint operation is triggered periodically by LFS.

Even though LFS provides scalable performance on disks, it cannot be directly used on flash memory, because it is designed only for disks. In particular, LFS violates one of the flash memory characteristics; it overwrites several areas such as super blocks and checkpoint area. In addition, LFS is not aware of flash memory and it does not actually erase its segments while it performs the segment cleaning operation.

Table I compares the characteristics of LFS, JFFS2, YAFFS, and ScaleFFS.

Table I. Comparisons among LFS, JFFS2, YAFFS, and ScaleFFS

File System	LFS	JFFS2	YAFFS	ScaleFFS
Target storage media	Hard disk	NOR (NAND)	NAND	NAND
Target storage size	<0.5GB	<64MB (max 4GB)	<64MB (max 1GB)	>32MB
Index structure	Inode (in disk)	Linked list (in memory)	Tree (in memory)	Inode (in flash)
Memory footprint	Small	Large	Large	Small
Mount time	Fast	Slow	Slow	Fast

3. DESIGN AND IMPLEMENTATION OF SCALEFFS

In this section, we present the basic design of ScaleFFS, a new scalable flash file system proposed for mobile multimedia systems with large-capacity NAND flash memory. ScaleFFS has two advantages over other flash file systems such as JFFS2 and YAFFS: (i) the memory space required by ScaleFFS is small and independent of the file system size, and (ii) the mount time in ScaleFFS is short and constant.

3.1 Design Goals

As presented in Section 2, traditional flash-aware file systems, especially JFFS2 and YAFFS, suffer from excessive memory consumption and long mount time. Memory consumption is one of the most important issues in designing mobile multimedia systems, because the amount of memory is directly related to the cost and the battery lifetime. The mount time can also significantly increase the system start-up time. Making this even more critical, both the memory consumption and the mount time are proportional to the storage capacity. ScaleFFS is proposed here to solve these problems, the main design goals of which are as follows.

—*Support for large-capacity NAND flash memory*. ScaleFFS can support more than tens of gigabytes of storage capacity composed of NAND flash memory. Individual files can be as large as several gigabytes.

—*Small memory footprint*. ScaleFFS does not keep file system information in memory permanently. Most metadata, such as directory entries and index structures, are easily locatable on flash media so that the file system can retrieve them on demand at runtime.

—*Fast mount*. Scanning the entire flash memory is not practical for large-capacity storage media. Even if the file system crashes, it can be recovered quickly to minimize the start up time.

—*Comparable performance*. ScaleFFS provides I/O throughput comparable to that of other flash file systems. The I/O performance should not be sacrificed even if the index structures are fetched from flash memory on demand. Thus, for ScaleFFS the overhead associated with reading or writing the index structures should be reduced.

3.2 ScaleFFS Basic Design

ScaleFFS is a log-structured file system whose metadata and directory structure are more similar to LFS [Rosenblum and Ousterhout 1992] than to JFFS2 or YAFFS. The structure of LFS is not only scalable to large storage capacity, but also suitable for flash memory. The entire indexes and directory structures of LFS do not have to be read into memory at mount time. In LFS, most information can be easily accessed from storage media on demand. In addition, the log structured architecture avoids in-place updates and facilitates a fast recovery operation. Therefore, the structure of LFS is a perfect

Table II. Extra Information Stored in Spare Area

Field	Size (bytes)
Next Log Block (First page only (cf. Sec. 4.3))	4
Bad Block Indicator	2
Block Type	1
Inode Number	4
Offset with its file	4
ECC	12
Last Modified Time	4
Reserved	1
Total	32

choice for achieving our design goals, and in this regard many philosophies and mechanisms from LFS were borrowed in designing ScaleFFS.

Although both JFFS2 and YAFFS also employ a log structure, they are not scalable, because they only adopt a sequential logging scheme, which writes data in a sequential manner, from LFS. They do not use inode-like or any other index structures that are easily locatable and accessible at runtime. Consequently, they need to scan the entire flash media to construct the index structures and to keep them in memory. This causes scalability problems such as slow mount time and large memory footprint.

While our basic design borrows many aspects from LFS, there are numerous differences between ScaleFFS and LFS, particularly between hard disks and flash memories. In the following, we highlight some of the unique features in ScaleFFS. First, the buffering scheme of ScaleFFS is different from that of LFS. LFS buffers all the modified data until its buffers are full or a checkpoint operation is initiated to maximize write throughput by eliminating disk seeks. In contrast, ScaleFFS attempts to write a block to flash memory immediately if the block is completely filled with new data. This is because, in flash memory, bulk writing provides little performance gain, but rather increases the latency of flushing.

Second, ScaleFFS utilizes spare areas of NAND flash memory to store extra information whenever it updates a block. For crash recovery and garbage collection, the block type, inode number of the file containing that block, the offset within the file, and the last modified time are stored in the spare area. Spare area also has an error-correcting code (ECC) for error correction.[1] Currently, ScaleFFS combines two 512-byte sectors as a single block (1KB), and uses associated 32 bytes of spare area to store these extra data. Table II summarizes the extra information stored in spare area. On the other hand, LFS is not aware of spare area and cannot utilize it, as it is designed only for hard disks.

Third, the segment size of ScaleFFS is different from that of LFS. In ScaleFFS, utilizing large size for a segment is useless, because there is little performance benefit from sequential bulk write operations. The segment size of ScaleFFS is the same as the flash memory erase block size while LFS uses much larger sizes.

Finally, the checkpoint region of ScaleFFS is designed to lengthen the lifetime of flash memory. LFS writes checkpoint data in a fixed position. This can lead to early wear-out of the checkpoint region in flash memory. In order to prevent wear-out of the checkpoint region, ScaleFFS allocates larger space for the checkpoint and uses smaller checkpoint data by employing *indirect_inode_map* blocks. In ScaleFFS, a checkpoint region is located at the end of the flash memory space, as shown in Figure 2. We present details of the checkpoint region in the next section.

[1]In this paper 1-bit ECC, which requires 3-byte codes for 512 byte data, is used. Currently, manufacturers recommend a 1-bit ECC for large block NAND flash [Samsung Electronics Co., Ltd. 2007, 2006, 2007].

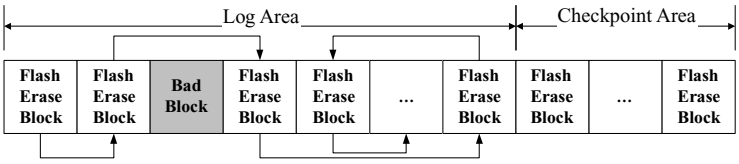

Fig. 2. The overall physical flash layout of ScaleFFS. ScaleFFS manages virtually sequential log space with spare areas. The checkpoint area is positioned at the fixed region, which is the end of the flash memory space.

4. DETAILED FILE SYSTEM OPERATIONS

In this section, we present detailed file system operations of ScaleFFS. Although the basic design of ScaleFFS originates from LFS, ScaleFFS is modified and optimized for NAND flash memory, owing to the unique characteristics of this type of flash memory.

4.1 Read and Write

To read a file, ScaleFFS first identifies the inode number of the file from directory entries. ScaleFFS then locates the inode block of the file by searching the inode map with the inode number. After locating the inode block, the file system reads the requested block. If the inode and indirect blocks are cached in memory, this process can be accelerated.

When ScaleFFS writes a file, it updates the modified block in a sequential fashion, in the same manner as LFS. ScaleFFS first writes data blocks of the file, and then writes the inode and/or indirect blocks if necessary. Inode map blocks are written during the checkpoint operation.

As described in the previous section, ScaleFFS will immediately flush modified data. ScaleFFS allocates a data buffer that can store a single data block (currently 1KB) and several metadata block buffers in memory. If the data buffer is full, ScaleFFS immediately writes the data in the buffer to flash memory.

After flushing data blocks, ScaleFFS updates the inode and indirect blocks. This is done only in the metadata buffers, since metadata are more likely to be updated again than are data blocks. When a kernel or user makes a request to sync file system data, ScaleFFS writes all modified metadata blocks. This approach can dramatically reduce the overhead caused by metadata updates.

Whenever ScaleFFS writes a block, the extra information described in Table II is simultaneously recorded in the spare area. The information is used during mount time to recover the file system from crashes and during garbage collection to reclaim obsolete blocks.

4.2 Directory Operations

Whenever directory operations such as creation, deletion, and rename take place in ScaleFFS, the corresponding directory block is updated. Since directory updates are less frequent than data updates in a mobile multimedia file system, directory updates are directly applied to flash memory. This makes it easier to recover directory structures in the case of power-off failure than caching and flushing the updates at a later time.

The aforementioned strategy is still not sufficient to make the file system consistent for several directory operations such as creation, deletion, and rename after power-off failure. These operations accompany multiple blocks to be updated. For instance, a file creation requires that both directory and inode blocks be updated, and a rename operation writes directory blocks with renamed entries followed by updating of directory blocks where the original entry was removed. In order to detect incomplete directory operations after power-off failure, each directory block contains directory update information.

This information includes the inode number of an updated file or a directory, and the type of update operation.

4.3 Free Log Management

Unlike LFS, which attemtps to secure large extents of free space to fully utilize disk bandwidth, ScaleFFS only maintains the log space in a virtually sequential order. The virtually sequential log is a log that can be accessed sequentially regardless of the physical order of the space. In ScaleFFS, each flash erase block is logically threaded similar to a linked list structure, as shown in Figure 2. This eases free space reclamation, since the flash erase block is the unit of erase operation. Note that the size of the flash erase block (128KB for large block NAND) is much smaller than that of the extent used in LFS, which is usually larger than 512KB.

ScaleFFS uses spare areas to manage free log space. ScaleFFS allocates a pointer to the next flash erase block in the spare area of the first page of each block (see Table II). If a free flash erase block is filled with newly written data, ScaleFFS simply follows the pointer and allocates new space in the next erase block. Initially, every flash erase block, except for the last one, has a pointer indicating the logically adjacent flash erase block. Once a flash erase block is erased, it is added to the end of the free log space by updating the spare area of the log.

4.4 Checkpoint

A checkpoint ensures that all the file system structures before the checkpoint are consistent and complete. ScaleFFS performs the checkpoint operation occasionally to keep the file system consistent. In ScaleFFS, the checkpoint operation involves two steps. First, it writes all modified data to the log, including the internal buffers, indirect blocks, inodes, and inode map blocks. Second, it writes the checkpoint information in the checkpoint region in a round-robin manner.

File system operations after the last checkpoint would be incomplete and inconsistent. For example, inode map blocks and indirect inode map blocks are not complete if the checkpoint is not triggered. These data must be scanned and handled during mounting of the file system. This can lengthen the mount time, and the time is dependent on the size of the log written after the last checkpoint.

In order to reduce the recovery time, ScaleFFS makes a checkpoint at every 4MB data written and at the unmount time. Since scanning the spare area of 4MB flash space takes about 221ms, ScaleFFS can bound the mount time even in the event of a file system crash.

Repeated checkpoint operations may wear out the flash erase blocks in the checkpoint region rapidly, thus eventually reducing the file system lifetime. Suppose that a user copies 1GB of multimedia files such as movie files or MP3 files everyday, and flash memory has a limit of 100,000 erase cycles. The checkpoint region is then overwritten 256 times a day, and the flash erase block containing the checkpoint data will be worn out in 391 days.

In order to lengthen the lifetime of the checkpoint region, we apply two simple approaches: allocating a larger flash memory space to the checkpoint region and compacting the amount of checkpoint data. In the current implementation, ScaleFFS uses 512KB for the checkpoint region and the size of checkpoint data is 1KB. Even if 4GB of data is written to the file system per day, the estimated lifetime of the checkpoint region will be 137 years according to Equation (1).

$$\begin{aligned} \text{Lifetime} &= \frac{\text{Erase limit} \times \text{Total checkpoint region size}}{\text{Checkpoint size per day}} \\ &= \frac{100,000 \times 512\text{KB}}{4\,(\text{GB/day})/4\text{MB} \times 1\text{KB}} \\ &= 50,000\text{days} \approx 137\text{years}. \end{aligned} \tag{1}$$

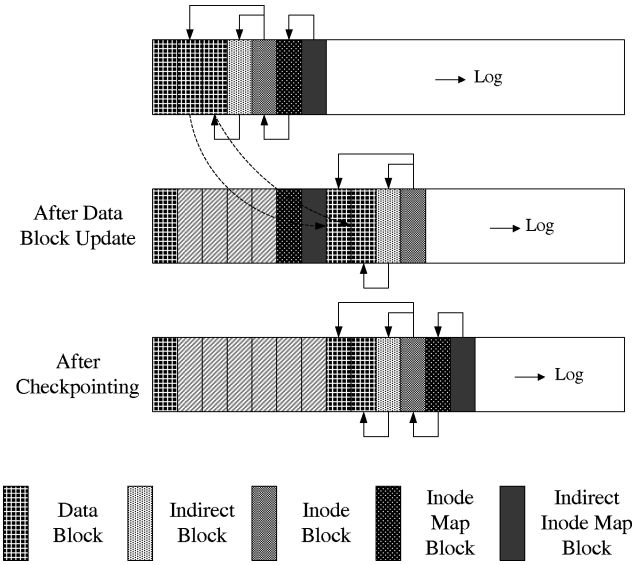

Fig. 3. Log structure of ScaleFFS. ScaleFFS writes only modified data blocks at the end of the log, and it flushes metadata during sync operations. *Indirect Inode Map Blocks* point to *Inode Map Blocks*.

Although the lifetime of flash storage varies depending on the behavior of the user, we can see that this configuration provides sufficient lifetime for a mobile system.

One problem with the checkpoint is that the size of checkpoint data to record all the locations of inode map blocks can exceed 1KB, if the number of files increases. In order to allow all checkpoint data to be able to fit into 1KB, ScaleFFS introduces an indirect inode map, as shown in Figure 3. The indirect inode map holds all the addresses of inode map blocks and ScaleFFS stores only the locations of the indirect inode map blocks in the checkpoint region. Therefore, using the indirect inode map blocks, we can find the location of an inode map block, and the inode map block in turn has the information to locate a specific inode. This additional indirection enables ScaleFFS to support several millions of files without increasing the size of checkpoint data. For example, if 128 indirect inode map blocks are used, ScaleFFS can store more than eight million files.[2] This number is large enough for mobile multimedia systems and even for normal desktop computers.

To summarize, the checkpoint region of ScaleFFS consists of the addresses of indirect inode map blocks, the position of the last written block in the log, and the last erase block to identify the end of the log. Table III presents the details of the checkpoint region, and the resulting layout of the log is illustrated in Figure 3.

4.5 Mount and Recovery

During the file system mount, ScaleFFS reads the last version of checkpoint data and uses it to initialize the file system. In order to find the last version of checkpoint data, ScaleFFS finds the last written checkpoint block by scanning the checkpoint region. Scanning the checkpoint region requires a two-phase operation. First, ScaleFFS scans the spare area of the first page in each block in the checkpoint region and finds the block with the most recent modified time. ScaleFFS then searches for the last

[2]ScaleFFS currently uses 1KB blocks. Each inode map block points to 256 inodes and an indirect inode map block points to 256 inode map blocks.

Table III. Summary of Checkpoint Region

Field	Size (byte)	Description
highest_inode	4	The highest inode number
nr_used_inodes	4	The number of used inodes
nr_valid_blocks	4	The number of live blocks
nr_free_blocks	4	The number of free blcks
nr_invalid_blocks	4	The number of dead blocks
nr_bad_blocks	4	The number of bad blocks
current_log_block	4	The position of the last written block
last_log_block	4	The position of the block of the free log
indirect_inode_map_blocks	512	The array of pointers to indirect inode map blocks
time	4	Timestamp
reserved	476	Reserved area for future use
Total	1024	

checkpoint data within that block by comparing the last modified time field in each spare area. This two-phase scan provides faster mount time than scanning all spare areas in the checkpoint region.

After reading the last checkpoint data, ScaleFFS reads all the indirect inode map blocks and stores them in its own buffer. While the maximum size of these blocks is 128KB, it is usually less than 16KB if the number of files does not exceed one million. Thus, reading and storing all the indirect inode map blocks rarely affects memory consumption and mount time.

In the event of a file system crash, ScaleFFS automatically detects the crash and runs a recovery procedure at the next mount. At the beginning of the file system mount, ScaleFFS looks up the latest checkpoint data and scans spare areas of the log after the last log position pointed to by the checkpoint data. Since ScaleFFS makes a checkpoint at each unmount, data after the last log position indicate that a file system crash has occurred.

The recovery procedure follows the log after the last checkpoint, and replays or undoes file system operations that are inconsistent or incomplete. For example, if a data block is written, but its inode is not, the block is discarded. On the other hand, if the inode is successfully written, write operations can be completed by recovering the inode map and indirect inode map. The inode map and indirect inode map are easily recovered if the inode is correctly written. For an incomplete directory operation, ScaleFFS makes the file system consistent by undoing the operation.

4.6 Garbage Collection

In our current implementation, ScaleFFS employs a round-robin garbage collection policy that is also adopted in the original JFFS and YAFFS. During garbage collection, ScaleFFS scans spare areas in a victim erase block and identifies whether each block is alive or dead by consulting the corresponding inode map and inode. Note that examining the inode map and inode takes little time, because they can be cached in memory, and the total amount of time to reclaim a block that has only invalid pages is 2.4ms when the inode map is cached.

5. EXPERIMENTAL RESULTS

5.1 Evaluation Methodology

We have implemented an initial version of ScaleFFS based on Linux kernel 2.6.15. We evaluated the performance of ScaleFFS on a PC with a Pentium IV processor and 512MB RAM. NAND flash memory was emulated by NANDSIM included in the Linux MTD (Memory Technology Device) package [MTD 1999]. NANDSIM is a virtual NAND flash memory chip that provides flash memory space using RAM

Table IV. The Characteristics of NANDSIM

Flash Memory Operation	Access Time (μs)
Read a page (512B)	55.7
Read a spare area (512B)	27.0
Write a page (512B)	237.7
Erase a block (16KB)	2005

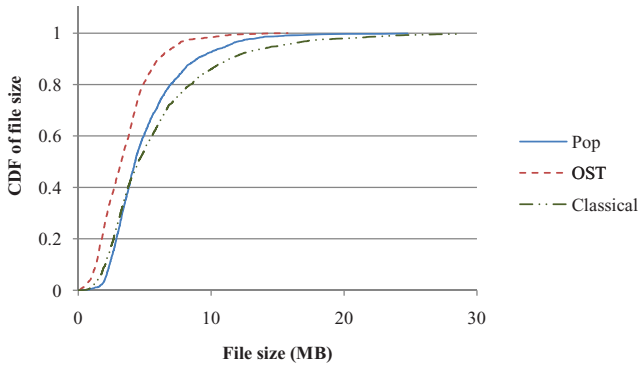

Fig. 4. Distribution of MP3 file sizes.

and emulates the latency of flash operations with busy waiting. In our evaluation, NANDSIM is modified to emulate the actual flash memory performance. The characteristics of NANDSIM are summarized in Table IV. During our experiments, we used 256MB flash memory emulated by NANDSIM.

The performance of ScaleFFS is compared with that of JFFS2 without data compression and YAFFS,[3] respectively. We primarily measured the mount time and the file system throughput. First, in order to evaluate the effect of file system size to the mount time, we repeated the mount operation several times changing both the total flash memory size and the amount of data stored in the file system. To fill each file system, 5MB MP3 files are used. Second, we ran a microbenchmark to measure the file system throughput. The microbenchmark evaluates the read and write performance of file systems by accessing MP3 files. It also measures random seek latency within a 100MB movie file. In addition, we analyze the memory consumption of each file system and the metadata overhead of ScaleFFS by counting the number of accessed blocks for each type (data, inode, indirect, etc.).

For microbenchmarks, we collected the characteristics of multimedia files from a user using a MP3 player. The audio files are classified into three genres, pop, classical music, and original sound track (OST). 92.6% of the pop audio clips and 98.3% of the OST audio clips are smaller than 10MB, and the average sizes of the pop and OST MP3 files are 5.2MB and 3.5MB, respectively. The maximum size of classical music files is 55MB, and the average size is 5.9MB. The distribution of file length is illustrated in Figure 4. In the microbenchmark, 25 pop music files, 15 OST music files, and 9 classical music files are selected according to the distribution.

5.2 Mount Time

Figure 5 shows the mount time with respect to the total capacity of NAND flash memory when the file system has no data. We can clearly see that ScaleFFS requires much shorter time to mount a file system and the time is almost constant, only seven milliseconds in our environment. This is because ScaleFFS only scans spare areas in the checkpoint region and indirect inode map blocks. Since the checkpoint

[3]The results for JFFS2 and YAFFS are obtained from actual implementations running on the Linux kernel 2.6.15.

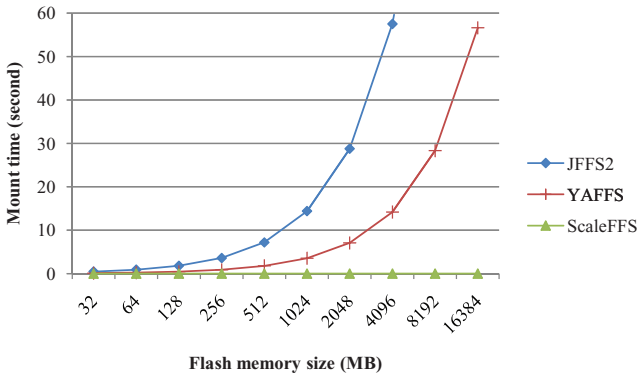

Fig. 5. Mount time with respect to NAND flash memory size.

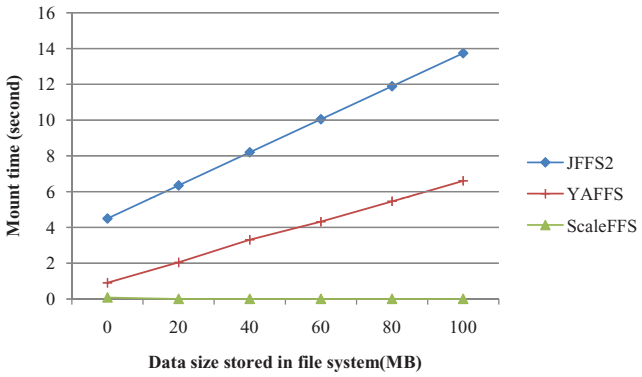

Fig. 6. The effect of data size on the mount time. (The total file system size is 256MB.)

region consists of a fixed number of flash erase blocks in ScaleFFS, the mount time remains constant regardless of the file system size. In contrast, the time to mount JFFS2 or YAFFS almost linearly increases as the capacity of flash memory grows. Even though no data are stored in the file system, JFFS2 scans the `JFFS2_NODETYPE_CLEANMARKER` node in each flash erase block, and YAFFS examines the first page of each flash erase block to find chunks. Note that the mount time results for file systems using NAND flash memory size larger than 256MB are estimated values based on the number of flash memory access to be scanned during mounting of each file system, because NANDSIM can only emulate NAND flash memory whose size is less than the available system memory size.

Figure 6 illustrates the mount time when the amount of stored data is varied and the total file system size is 256MB. The results present similar patterns, as can be seen in Figure 5. Since ScaleFFS scans a limited number of flash erase blocks, the results are always the same for all conditions. However, JFFS2 and YAFFS again take much longer time, in proportion to the amount of stored data.

The mount time is also affected by the number of files in the file system. Figure 7 shows the mount time according to the number of zero-size files in each file system. The mount time of ScaleFFS remains almost constant while JFFS2 and YAFFS require longer time to start up as the number of files increases. Because one indirect inode map block in ScaleFFS can cover up to 65536 files, ScaleFFS only needs

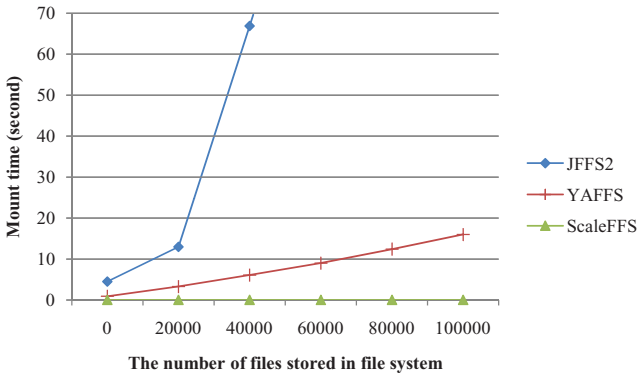

Fig. 7. The effect of the number of files on the mount time. (The total file system size is 256MB.)

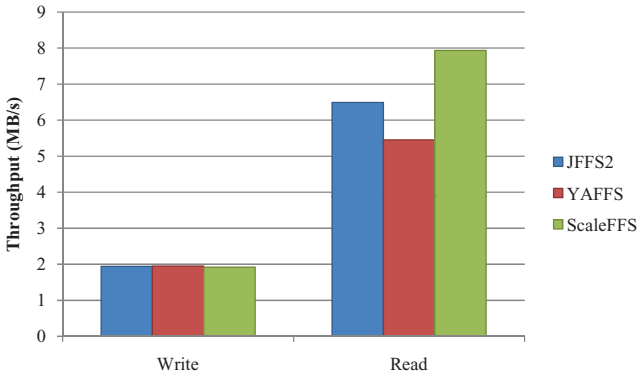

Fig. 8. The performance of read and write operations.

to scan two indirect inode map blocks to handle 100,000 files. On the other hand, JFFS2 and YAFFS require time consuming scanning operations to retrieve metadata of all files from flash memory. The latency to mount JFFS2 or YAFFS is a linear function of the number of nodes or chunks in flash memory. Typically, the number of nodes or chunks is directly related to the number of files and directories, the number of flash erase blocks, and file sizes.

From Figures 5, 6, and 7, we can observe that, in terms of the mount time, ScaleFFS is scalable to large-capacity NAND flash memory. Long mount time of up to several tens of seconds makes JFFS2 and YAFFS impractical for portable devices equipped with a few gigabytes of NAND flash memory.

5.3 File System Throughput

The microbenchmark results are depicted in Figure 8, where “read” and “write” represent the throughput of read and write operations, respectively. For the write throughput, the microbenchmark measures the latency to write 155MB of MP3 files to each file system that initially has no data. Read throughput is measured by reading all the MP3 files from flash memory.

ScaleFFS, JFFS2, and YAFFS show almost the same write bandwidth. The three file systems utilize more than 93% of the write bandwidth of NANDSIM (2.05MB/sec). The slight difference in the write performance comes from the additional cost of each file system. For example, ScaleFFS reads spare area to get the next free block and flushes metadata block and checkpoint data. In YAFFS, there are read

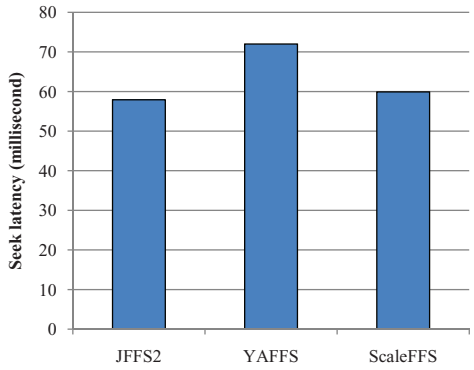

Fig. 9. Random seek latency.

operations to test whether a block is erased or not, because YAFFS does not use a mark that represents a block is erased. During the benchmark, it was measured that ScaleFFS performs 0.6% additional write operations compared to YAFFS for managing inode blocks, indirect blocks, inode map blocks, and checkpoint blocks. In addition, ScaleFFS reads spare area to find the next free block whenever a block is fully filled with newly updated data. In JFFS2, there are no extra operations except for writing node headers. Since data is not aligned to the size of the page due to the node header, JFFS2 performs 2% more write operations than YAFFS. Because writing data is the dominant component of write time, this overhead is tolerable for a large-capacity flash file system.

The read performance, however, varies substantially among the three file systems. ScaleFFS shows about 22% and 45% higher read throughput than JFFS2 and YAFFS, respectively. Once inode and indirect blocks are read into the kernel buffer cache, they are rarely removed from the cache, because they are frequently accessed. Thus, ScaleFFS can utilize the full read bandwidth in transferring file data. According to the results, ScaleFFS provides 7.9MB/s read bandwidth, which is enough to play a movie file that is more than 1Mbps. In JFFS2, however, all node headers of a file first need to be scanned to build detailed index structures in memory from the minimal in-memory data structure before reading the actual file data (cf. Section 2.2). This procedure makes JFFS2 scan the pages containing the headers twice. Moreover, if a node is not aligned to the NAND flash memory page boundary due to its header, there is an additional overhead to read more pages. As a result, JFFS2 can utilize only 74.1% of the read bandwidth of NANDSIM (8.77MB/sec).

YAFFS shows much lower read performance than the others. Since YAFFS maintains 16-bit index information in memory, it can only locate 65,536 pages. If flash memory has more than 65,536 pages, YAFFS groups several pages and uses 16-bit addresses to find a group. YAFFS then scans spare areas within the group to find the exact page. For this reason, the read performance of YAFFS is degraded according to the flash memory size.

Figure 9 compares the random seek performance of ScaleFFS, JFFS2, and YAFFS. The average latency is measured by reading 4KB data blocks at random offsets from a 100MB movie clip. The average latency of JFFS2 is about 3.3% shorter than that of ScaleFFS. This is because ScaleFFS reads index structures (inode and/or indirect blocks) from flash memory on demand, while JFFS2 exploits in-memory structures to find data. However, JFFS2 should build the in-memory structures for a fast seek when it opens a file. This takes about 14.6 seconds for a 100MB file. Therefore, JFFS2 is not suitable for handling large multimedia files. The random seek latency of YAFFS is more than 20% longer than other file systems, due to the same factors described above that led to degraded read performance.

Table V. Summary of Memory Consumption in ScaleFFS, JFFS2, and YAFFS

	Memory Consumption		
Component/Scenario	ScaleFFS (maximum value)	JFFS2	YAFFS
Checkpoint data and super block	1KB	0.3KB	3.6KB
Write buffer	1KB	0.5KB	5KB
Indirect inode map	128KB	N/A	N/A
Metadata cached during writing a 10MB file	43KB	185KB	46.9KB
Metadata cached during writing a 100MB file	410KB	1.8MB	459.3KB
Metadata cached during writing a 1GB file	<4.5MB (estimated)	>18MB (estimated)	<4.5MB (estimated)

5.4 Memory Consumption

One of our design goals is to reduce the memory consumption even if a large-capacity flash memory is used. In order to achieve this goal, ScaleFFS only keeps minimal information in memory such as the indirect inode map and the checkpoint data. The indirect inode map occupies at most 128KB when the file system has eight million files, and the checkpoint data requires only 1KB. Thus, operation of ScaleFFS requires limited and almost constant memory space.

During file system operations, various metadata and file data are cached temporarily in memory for better performance. Cached data could be evicted at any time except for modified metadata. Even though the total size of metadata in ScaleFFS is proportional to the size of data, metadata does not need to be kept in memory. Thus, the amount of modified metadata is of no consequence in ScaleFFS. Table V summarizes the memory consumed by ScaleFFS.

On the other hand, JFFS2 permanently keeps all metadata of files in memory while the files are being used by a user or the kernel. If JFFS2 reads or writes a 1GB movie file, it requires at least 18MB of memory space, as shown in Table V. Moreover, JFFS2 needs 16-byte additional memory space for each data node stored in flash memory media. YAFFS consumes two bytes for each chunk (page) and 124 bytes for each file even if a file is not opened to be accessed. The total metadata size needed by YAFFS to write a 1GB file appears to be similar to that in the case of ScaleFFS (cf. Table V). However, cached metadata in ScaleFFS can be evicted if the memory space is not sufficient whereas this cannot be done in YAFFS. Thus, both JFFS2 and YAFFS suffer from significant memory footprint size when applied to large-scale flash memory storage and multimedia data. This memory problem of JFFS2 and YAFFS is exacerbated as the size of the flash memory and data grow.

5.5 Metadata Overhead

Unlike JFFS2 or YAFFS, ScaleFFS stores metadata and file data separately, and hence it requires additional write or read operations to update or access metadata blocks. Nevertheless, the performance of ScaleFFS is comparable to that of JFFS2 or YAFFS, because these metadata blocks are cached and updated in the memory cache.

Table VI summarizes the percentage of time taken to access each block type for three scenarios. The scenario denoted as “Downloading MP3 files” models copying MP3 files from a host to a player. The second scenario, labeled “Playing MP3 files,” is a workload produced by a media player in Linux that selects MP3 files in random order from its play list and plays them. The player first reads header information of MP3 files in its player list, and then reads audio data in a file sequentially. The last scenario (“Random seek test”) is the random seek test presented in Section 5.3.

If a file is accessed sequentially, the hit ratio of the metadata is high, and thus the overhead for accessing metadata is negligible. This means that the overhead is not significant for multimedia applications that exhibit sequential file I/O patterns. In contrast, if files are small, the portion for accessing

Table VI. Percentage of Time Taken to Access Each Block Type in ScaleFFS

File System Block Type	Workloads		
	Downloading MP3 Files	Playings MP3 Files	Random Seek Test
Data block	99.40%	99.40%	80.2%
Inode block	0.04%	0.01%	0.4%
Indirect block	0.47%	0.58%	18.8%
Inode map block	0.03%	<0.01%	0.2%
Indirect inode map	0.03%	<0.01%	0.2%
Checkpoint block	0.03%	<0.01%	0.2%

metadata becomes significant, because ScaleFFS needs to access an inode map block and an inode block to read or to write a file.

The metadata overhead during the random seek test is relatively higher than sequential workloads. This results from the aspect that if the memory cache does not contain needed metadata, ScaleFFS requires at most three flash memory accesses to find the location of a data block. However, as already seen in Figure 9, this does not significantly degrade the overall performance since the metadata are rarely evicted from the cache once they are read from flash memory.

6. CONCLUSIONS AND FUTURE WORK

This paper presented the design and implementation of ScaleFFS, a scalable flash file system for mobile multimedia systems equipped with large-capacity NAND flash memory. The structure of ScaleFFS was originally inspired by LFS, but is significantly modified and optimized for NAND flash memory. In particular, ScaleFFS exploits spare areas of NAND flash memory to store extra information and flushes data as soon as possible. In addition, ScaleFFS does not maintain large extents of free space, and its checkpoint mechanism is designed to provide long lifetime with flash memory.

Our measurement results have shown that ScaleFFS requires only a small fixed amount of time to mount a file system. In terms of file system performance, ScaleFFS shows almost the same write bandwidth and up to 45% higher read bandwidth relative to JFFS2 and YAFFS. Given its small memory footprint and fast mount time, combined with higher performance than the existing flash file systems, ScaleFFS is an ideal choice for large-scale NAND flash memory.

For our current implementation, it has been confirmed that the index structure of a disk-based file system is also applicable for NAND flash memory. The overhead to manage the index structure is found to be acceptable. In the near future, we plan to redesign the garbage collection of ScaleFFS to achieve more efficient flash memory utilization and better I/O performance.

REFERENCES

AlephOne Ltd. 2003. *Yet Another Flash Filing System (YAFFS)*. Aleph One Limited. http://www.aleph1.co.uk/yaffs.

Card, R., Ts'o, T., and Tweedie, S. 1994. Design and implementation of the second extended filesystem. In *Proceedings to the 1st Dutch International Symposium on Linux*. State University of Groningen, Groningen, Netherlands.

Douglis, F., Caceres, R., Kaashoek, M. F., Li, K., Marsh, B., and Tauber, J. A. 1994. Storage alternatives for mobile computers. In *Proceedings of the Conference on Operating Systems Design and Implementation*. 25–37.

Gummadi, K. P., Dunn, R. J., Saroiu, S., Gribble, S. D., Levy, H. M., and Zahorjan, J. 2003. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM Press, New York, NY, 314–329.

Huang, H., Pillai, P., and Shin, K. G. 2003. Design and implementation of power-aware virtual memory. In *Proceedings of USENIX Annual Technical Conference*. 57–70.

Hynix Semiconductor Inc. 2006. Hynix NAND flash data sheet. http://www.hynix.com/datasheet/pdf/flash/HY27UH08AG(5_D)M(Rev0.6).pdf.

KANG, J.-U., JO, H., KIM, J.-S., AND LEE, J. 2006. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT'06)*. ACM Press, 161–170.

KIM, J., KIM, J. M., NOH, S. H., MIN, S. L., AND CHO, Y. 2002. A space-efficient flash translation layer for compactflash systems. *IEEE Trans. Consum. Electron. 48,* 2 (May), 366–375.

LIM, S.-H. AND PARK, K.-H. 2006. An efficient NAND flash file system for flash memory storage. *IEEE Trans. Comput. 55*, 7, 906–912.

MARSH, B., DOUGLIS, F., AND KRISHNAN, P. 1994. Flash memory file caching for mobile computers. In *Proceedings of The 27th Hawaii International Conference on Systems Science*. 451–460.

MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. 1984. A fast file system for UNIX. *ACM Trans. Comput. Syst. 2*, 3, 181–197.

MTD 1999. Memory Technology Device (MTD) subsystem for Linux. `http://www.linux-mtd.infradead.org/index.html`.

PARK, C., KANG, J.-U., PARK, S.-Y., AND KIM, J.-S. 2004. Energy-aware demand paging on NAND flash-based embedded storages. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design (ISLPED'04)*. ACM Press, New York, NY, 338–343.

PAULSON, L. D. 2005. Will hard drives finally stop shrinking? *IEEE Comput. 38*, 6, 14–17.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1, 26–52.

SAMSUNG ELECTRONICS CO., Ltd. 2007. Samsung NAND flash data sheet. `http://www.samsung.com/products/semiconductor/NANDFlash/SLC_LargeBlock/32Gbit/K9NBG08U5A/K9NBG08U5A.htm`.

SAROIU, S., GUMMADI, K. P., DUNN, R. J., GRIBBLE, S. D., AND LEVY, H. M. 2002. An analysis of internet content delivery systems. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*. ACM Press, New York, NY, 315–327.

SELTZER, M. I., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. 1993. An implementation of a log-structured file system for UNIX. In *Proceedings of USENIX Winter Technical Conference*. 307–326.

SIVATHANU, M., BAIRAVASUNDARAM, L. N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2004. Life or death at block-level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*. 379–394.

STMicroelectronics 2007. `http://www.st.com/stonline/products/literature/an/10123.pdf`. STMicroelectronics ECC in SLC NAND flash.

WOODHOUSE, D. 2001. JFFS : The journalling flash file system. presented in the Ottawa Linux Symposium.

Received November 2006; revised June 2007; accepted September 2007