FlashLight: A Lightweight Flash File System for Embedded Systems

JAEGEUK KIM, HYOTAEK SHIM, SEON-YEONG PARK, and SEUNGRYOUL MAENG,
Korea Advanced Institute of Science and Technology
JIN-SOO KIM, Sungkyunkwan University

A very promising approach for using NAND flash memory as a storage medium is a flash file system. In order to design a higher-performance flash file system, two issues should be considered carefully. One issue is the design of an efficient index structure that contains the locations of both files and data in the flash memory. For large-capacity storage, the index structure must be stored in the flash memory to realize low memory consumption; however, this may degrade the system performance. The other issue is the design of a novel garbage collection (GC) scheme that reclaims obsolete pages. This scheme can induce considerable additional read and write operations while identifying and migrating valid pages. In this article, we present a novel flash file system that has the following features: (*i*) a lightweight index structure that introduces the *hybrid indexing scheme* and *intra-inode index logging*, and (*ii*) an efficient GC scheme that adopts a dirty list with an on-demand GC approach as well as fine-grained data separation and *erase-unit data allocation*. We implemented FlashLight in a Linux OS with kernel version 2.6.21 on an embedded device. The experimental results obtained using several benchmark programs confirm that FlashLight improves the performance by up to 27.4% over UBIFS by alleviating index management and GC overheads by up to 33.8%.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*Directory structures and file organization*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; B.3.2 [**Memory Structures**]: Design Styles—*Mass storage (e.g., magnetic, optical, RAID)*

General Terms: Design, Performance

Additional Key Words and Phrases: NAND flash memory, flash file system, index structure, garbage collection

ACM Reference Format:
Kim, J., Shim, H., Park, S.-Y., Maeng, S., and Kim, J.-S. 2012. FlashLight: A lightweight flash file system for embedded systems. ACM Trans. Embed. Comput. Syst. 11S, 1, Article 18 (June 2012), 23 pages.
DOI = 10.1145/2180887.2180895 http://doi.acm.org/10.1145/2180887.2180895

1. INTRODUCTION

Embedded systems such as MP3 players, cellular phones, personal digital assistants (PDAs), digital still cameras (DSCs), and portable media players (PMPs) constitute a major fraction of the digital systems market. NAND flash memory is widely used as a storage medium in embedded systems because of its advantageous features such as

This work was supported by the IT R&D Program of MKE/KEIT (2010-KI002090, Developmment of Technology Base for Trustworthy Computing).
Authors' addresses: J. Kim, H. Shim, S.-Y. Park, and S. Maeng, Computer Science Department, KAIST, Daejeon 305-701, Republic of Korea; email: {jgkim, htsim, parksy, maeng}@camars.kaist.ac.kr; J.-S. Kim, School of Information and Communication Engineering, Sungkyunkwan University, Suwon 440-746, Republic of Korea; email: jinsookim@skku.edu.

DOI 10.1145/2180887.2180895 http://doi.acm.org/10.1145/2180887.2180895

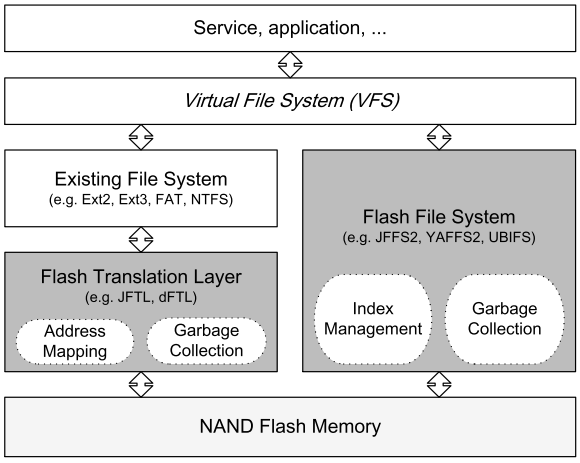


Fig. 1. Two major approaches for NAND flash-based storages.

small and lightweight form factor, solid-state reliability, and low power consumption [Douglis et al. 1994]. It has recently found even greater use due to its increased storage capacity as embedded systems require a large amount of secondary storage space with the ever-increasing requirement of capacity for storing multimedia contents.

NAND flash memory has several characteristics that differ from magnetic disks, which have been one of the most commonly used secondary storage devices. For example, it does not allow in-place updates, implying that previous data cannot be overwritten at the same location without being erased first. In addition, the erase unit, which we call the flash erase block (FEB), is relatively larger than the unit of read and write operations called *page*. Because of such differences, NAND flash memory cannot be directly applied to existing disk-based file systems.

To overcome these limitations, two major approaches have been proposed, as illustrated in Figure 1. One approach is to provide the Flash Translation Layer (FTL) between the existing file systems and flash memory [Choi et al. 2009; Kim et al. 2002]. The main purpose of FTL is to emulate the functionality of a block device with flash memory by hiding the *erase-before-write* characteristic as much as possible. Once FTL is available on top of the flash memory, any disk-based file system can be used. However, because FTL operates at the block device level, it is inaccessible to file system-level information like the liveness of data [Sivathanu et al. 2004], and this may limit the storage performance.

The other approach is to use *flash file systems* specially designed for flash memory. Over the past few years, several flash file systems have been studied and developed [Aleph One Ltd. 2003; Gal and Toledo 2005; Hunter 2008; Lim and Park 2006; Woodhouse 2001].

In order to design a flash file system with better performance, two issues should be considered carefully, as emphasized in previous literatures [Bityutskiy 2005; Chang et al. 2004]. One issue is to design an index structure that locates files and data stored in the flash memory. In early flash file systems such as JFFS2 [Woodhouse 2001] and YAFFS2 [Aleph One Ltd. 2003], the entire index structure was managed in the main memory. In large-capacity storage, however, many files occupying a large volume of space can be created and written in practice [Agrawal et al. 2007], making them suffer from high memory consumption. UBIFS [Hunter 2008], a recently proposed flash file system, solves this problem by fetching only the required indices on demand from the

flash memory; however, its performance is inferior to the in-memory approach used in JFFS2 and YAFFS2. In this article, we aim to design an on-flash index structure that performs better than UBIFS and comparable to JFFS2 and YAFFS2.

The other issue is to efficiently reclaim a number of scattered and invalidated pages produced by out-of-place updates. The process of reclaiming obsolete pages is called *Garbage Collection* (GC) [Lim and Park 2006], and it involves the following three steps.

(1) A proper victim FEB is selected among nonempty FEBs.
(2) In the victim FEB, valid pages are identified and copied to a new free FEB.
(3) The victim FEB is then erased.

The second step is called *valid page migration*, which produces most of the additional read and write operations during GC [Chang et al. 2004]. To reduce the valid page migration overhead, the GC scheme must identify valid pages instantly and reduce the number of valid pages that need to be copied. Previous flash file systems have made some efforts to mitigate the GC overhead; however, they have mainly focused on how to find victims and migrate valid pages, not on how to do the job faster and with less overheads. Both YAFFS2 and UBIFS may require a long time to identify valid pages because they need to read obsolete pages as well. Therefore, we aim to focus on enhancing the GC performance in the file system design level.

In this article, we present FlashLight, a novel lightweight flash file system that achieves high performance with the following features: (i) a lightweight index structure that uses a *hybrid indexing scheme* and *intra-inode index logging* to reduce the number of indirect indices that cause recursive index updates, and (ii) an efficient GC scheme that not only identifies valid pages instantly but also adopts a fine-grained data separation and *erase-unit data allocation* to reduce the number of valid pages that need to be copied. We implemented FlashLight on the NOKIA N810 platform [Nokia 2008] running the Linux kernel version 2.6.21. We evaluated FlashLight with three flash file systems, JFFS2, YAFFS2, and UBIFS, which are widely used in embedded systems.

Our contributions in this article can be summarized as follows.

(1) *Index Structure*. The previous flash file systems focused on how to translate inode numbers and data offsets into the physical locations of inodes and data in the flash memory. As a general solution, they adopt globally managed index structures such as in-memory chains and B+tree. This approach addresses the wandering tree problem[1], but makes the file system to traverse the index structure whenever looking up the inodes or data. Instead, we propose a *locally* managed scheme. We focus on how to efficiently use the user-created directory tree showing the directory locality, instead of adding another complex data structures (e.g., B+tree). In our approach, inodes are obtained directly from their parent inode that has the child indices while mitigating the wandering tree problem.
(2) *Garbage Collection*. Generally, GC policies are classified into two approaches: *passive* and *aggressive*. In the passive approach, which is adopted in JFFS2 and YAFFS2, all the data are written to the flash memory without any consideration of the data type, and GCs are performed with victims having the smallest migration overhead. This policy may limit the storage performance because hot and cold data are mixed together[2]. On the other hand, in the aggressive approach adopted in UBIFS and FlashLight, the data structures used in the file system are separated to different FEBs according to their hotness. UBIFS broadly separates metadata,

[1]This problem is introduced in Section 3.1.
[2]The effect of separating hot and cold data is discussed in Section 3.2.

Table I. Two Types of NAND Technology

	Single-Level Cell	Multi-Level Cell
Page Size (Bytes)	2,048	4,096
# of pages in an FEB	64	128
Spare Area Size (Bytes)	64	128
Read latency (μs)	77.8	165.6
Write latency (μs)	252.8	905.8
Erase latency (ms)	1.5	1.5

data, and index areas, while FlashLight divides the metadata area also into four independent areas such as DirInode, hash map, FileInode, and extent map[3].

The rest of this article is organized as follows. We present the background and related work in Section 2. Section 3 describes our motivations and Section 4 describes the design and implementation of the proposed flash file system. Section 5 presents the performance evaluation results. Finally, we conclude the article in Section 6.

2. BACKGROUND AND RELATED WORK

2.1 Background

A NAND flash memory chip consists of a set of blocks called FEBs (flash erase blocks), and each FEB contains a number of pages. A page is a unit of read and write operations, and an FEB is a unit of erase operation. Additionally, each page has *spare area* that is typically used to store error correction code (ECC) and other bookkeeping information.

There exist two types of NAND flash memory: Single-Level Cell (SLC) [Samsung Electronics] and Multi-Level Cell (MLC) [Samsung Electronics]. Table I lists the general specifications of the representative NAND chips. Note that the read/write latency shown in Table I includes the data transfer time between host and NAND flash memory.

A few bytes (typically 12∼16 bytes) of the spare area are assigned to ECC in SLC NAND chips. For MLC NAND chips, almost entire spare area needs to be allocated to ECC due to the high bit error rate (BER) of memory cells. In both types of chips, the number of write/erase cycles is strictly limited to 10,000∼1,000,000 times. This necessitates a *wear-leveling* process that aims at distributing the incoming writes evenly across the flash memory for a longer lifetime.

Recently, OneNAND flash memory was introduced to support both code and storage regions in a single chip [Samsung Electronics]. This fusion memory consists of SLC flash memory, buffer RAMs, ECC hardware, and other control logics. All the data as well as the code image are stored in the SLC flash memory, and the code area is typically 1,024 bytes long supporting eXecute-In-Place (XIP). In order to improve the performance of I/O operations, two page-sized buffer RAMs are interleaved one after the other. Because of these characteristics, OneNAND is widely used in embedded systems.

[3]During the Filebench test described in Section 5, FileInode pages were more frequently updated by approximately ten times than DirInode pages, and FlashLight reduced the number of migrated valid pages from 2,108 to 0 during GCs by dividing the metadata area.

2.2 Existing Flash File Systems

2.2.1 JFFS2 (Journaling Flash File System, v2). JFFS2 is a log-structured flash file system designed for small-scale embedded systems [Woodhouse 2001]. Originally, it was developed for NOR flash memory, but later extended to NAND flash memory.

In JFFS2, a node, which occupies a variable number of pages, is written sequentially to a free FEB. Nodes are typically categorized into three types: (1) INODE, (2) DIRENT, and (3) CLEANMARKER. Each INODE node contains the metadata of a directory or a file. A directory has one INODE node and several DIRENT nodes that contain directory entries. A file has a number of INODE nodes each of which contains a range of file data. When an FEB is erased successfully, JFFS2 writes a CLEANMARKER node to the FEB so that it can later be reused safely.

In the main memory, JFFS2 maintains a chained list for all the nodes including obsolete nodes. Each in-memory node consists of a physical address, node length, and pointers to the next in-memory nodes that belong to the same file. The memory footprint increases in proportion to the number of nodes, and this is a severe problem in large-capacity storage.

For managing FEBs, JFFS2 adopts additional in-memory linked lists: (i) the *clean list* of FEBs having only valid nodes, (ii) the *dirty list* of FEBs that contain at least one obsolete node, and (iii) the *free list* of FEBs that contain only CLEANMARKER nodes. From the dirty list, JFFS2 selects a victim FEB for GC, and checks for cross-references between in-memory nodes related to the victim FEB to identify and move the valid nodes. Note that, to handle wear-leveling and power-off recovery, JFFS2 also adopts *erasable list*, *bad list*, and so on.

Another problem of JFFS2 is a long mount delay. During the mount time, JFFS2 scans the entire flash memory to build the index structure in the main memory; this step takes from several to tens of seconds depending on the number of nodes.

2.2.2 YAFFS2 (Yet Another Flash File System, v2). YAFFS2 is another log-structured flash file system that was designed for NAND flash memory [Aleph One Ltd. 2003]. Similar to JFFS2, a *chunk* consisting of a set of pages and their spare ares is written sequentially to a free FEB in YAFFS2. The spare area in each chunk contains (i) the file ID that denotes the file inode number, (ii) the chunk ID that indicates the offset of the file data, and (iii) the sequence number that is incremented when a new FEB is allocated to find the up-to-date valid data after system reboot.

In the main memory, YAFFS2 stores the entire directory tree comprising a number of *objects* each of which represents a directory or a file. An object holds the physical location of its chunk in the flash memory, and it points to the parent and sibling objects. If an object is a directory, it also points to child objects. If an object is a file, a tree structure called *Tnode* is formed to provide the mapping from a file offset to the physical address of its chunk in the flash memory. In order to build these in-memory structures, YAFFS2 also suffers from large memory consumption similar to JFFS2.

For GC, YAFFS2 selects a suitable victim FEB, and identifies valid chunks by reading their spare areas. To reduce the mount delay, YAFFS2 adopts *checkpoint*, a well-known technique for fast system boot by reading a small amount of information.

2.2.3 UBIFS. UBIFS is designed for large-capacity storage by addressing the memory consumption problem in JFFS2 and YAFFS2 [Hunter 2008]. The key feature of UBIFS is to organize the index structure in the flash memory, whereas JFFS2 and YAFFS2 maintain it in the main memory.

In the flash memory, UBIFS adopts the node structure used in JFFS2 and a B+tree to manage the node indices. For the B+tree, two additional node types, MASTER and INDEX, are introduced in addition to those (INODE, DIRENT, and CLEANMARKER)

Table II. A Comparison of Flash File Systems

	JFFS2	YAFFS2	UBIFS	FlashLight
Storage Capacity	Small	Small	Large	Large
Memory Footprint	Large	Large	Small	Small
Mount Time	Long	Short	Short	Short
In-Memory Structure	Chained list	Dir-tree	TNC	Inode cache
On-Flash Structure	—	—	B+tree	Hybrid structure
Valid Page Identification	In-memory	On-flash	On-flash	In-memory
Data Separation Granularity	Coarse-grained	Coarse-grained	Fine-grained	More fine-grained

used in JFFS2. The MASTER node points to the root of the tree, and the leaves of the tree contain valid data. The internal elements of the tree are INDEX nodes that contain only pointers to their children.

When a leaf node is added or replaced, all the nodes from the parent INDEX node to the MASTER node must also be replaced. Updating all the ancestor INDEX nodes every time a new leaf node is written is very inefficient because almost the same INDEX nodes are written repeatedly. To reduce the frequency of updates, UBIFS defines a *journal*; it first writes a predefined number of leaf nodes to the journal instead of immediately inserting them into the B+tree. When the journal is considered full, the tree is reorganized with the leaf nodes in the journal.

For managing free FEBs, UBIFS adopts *LEB Properties Tree* (LPT). When the journal runs out of space, UBIFS searches for the LPT and takes a free FEB. When there are insufficient free FEBs, a GC process is triggered as follows.

(1) A suitable victim FEB is selected from the *dirty list*, which has the same structure as in JFFS2.
(2) All the nodes in the victim, including obsolete nodes, are read to check their validities.
(3) The valid nodes are moved to a free journal area, and the victim is erased.

To reduce the number of valid nodes that need to be moved, UBIFS separates the metadata, data, and index nodes to different FEBs, which has been proven to be efficient in the previous study [Lim and Park 2006].

In the main memory, UBIFS adopts *tree node cache* (TNC) to make tree operations more efficient by caching some INDEX nodes. To reduce the mount delay, UBIFS also adopts checkpoint, in a manner similar to YAFFS2.

3. MOTIVATION

3.1 Index Structure

To cope with the *erase-before-write* characteristic, many flash file systems employ a strategy in which the new data are written into an empty space, and the original data are invalidated. Due to this out-of-place update scheme, the physical location of data changes whenever it is overwritten, and accordingly, an index structure is required to store and keep track of the latest locations of the data.

As summarized in Table II, JFFS2 and YAFFS2 retain the complete index structure in the main memory. If there is insufficient memory, the mount fails. To reduce the memory footprint, UBIFS fetches only the required indices on demand from the flash memory, as most disk-based file systems do. The fetched indices are cached in the main memory for a while to accelerate subsequent accesses to the same indices, and they are eventually discarded later if the available memory becomes low. This on-demand scheme, however, results in a long latency for storing and retrieving indices during file

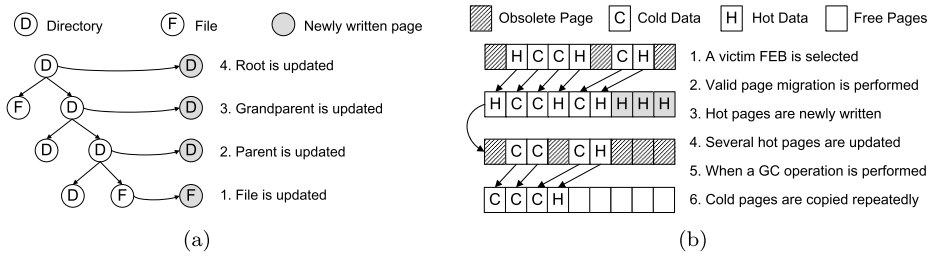


Fig. 2. Two issues in designing a flash file system: (a) the wandering tree problem and (b) the mixed hot and cold data problem.

system operations, and accordingly, it is essential to design an efficient on-flash index structure.

To design an efficient on-flash index structure, the *wandering tree problem* must be handled carefully [Bityutskiy 2005]. As illustrated in Figure 2(a), if a certain file in a directory tree is updated, the modified file is written in a newly allocated page. Because the pointer of the leaf file is now changed, the direct parent directory also needs to be updated. This necessitates another update in the grandparent directory, and eventually, updates are propagated to the root directory. Such recursive index updates should be minimized in order to avoid many costly write operations.

UBIFS adopts a B+tree for the on-flash index structure and a *journal* to mitigate the index management overhead. Nevertheless, while creating and deleting many small files, the index management overhead still ranges from 6.68% to 14.54% over the total elapsed time for Postmark and Filebench workloads (cf. Section 5). This overhead was mainly caused by managing the B+tree that stores all the metadata and data indices together.

3.2 Garbage Collection

Another important issue in designing a flash file system is the GC scheme. During GCs, additional read and write operations are induced by *valid page migration* that includes identifying and copying valid pages to other free FEBs. If a long time is required to perform valid page migration, the performance may degrade significantly [Chang et al. 2004].

To identify valid pages, the metadata can be stored in line with the file data on the flash memory; this is called a *node* in JFFS2 and UBIFS, and a *chunk* in YAFFS2. The metadata usually stores such information as the type of the data and the file it belongs to. By reading this metadata, the garbage collector can determine what pages need to be copied and what can be discarded. The downside of this approach is, however, that obsolete nodes are required to be read as well, thus degrading the performance. Alternatively, a new data structure can be adopted to reduce the overhead.

To reduce the number of valid pages that need to be copied, the garbage collector must avoid selecting a victim FEB with many hot pages; a hot page is one that includes data with a high probability of being updated and invalidated in the near future. As illustrated in Figure 2(b), if hot and cold data are mixed in an FEB, the cold data have a high chance of remaining valid at the next GC time, and thus, it repeatedly causes a considerable migration overhead. Therefore, it is necessary to store data in different FEBs according to their *hotness*. If the data are separated immaturely, it results in considerable degradation of the system performance [Chang et al. 2004].

As summarized in Table II, UBIFS also reads obsolete pages to check their invalidation, and separates metadata, data, and index nodes to reduce the number of valid pages that need to be copied. Our evaluation with Postmark and Filebench workloads

Table III. Summary of Major Log Areas on Flash Memory

Name	Contents	# of FEBs	Section
Checkpoint	File system information, a dirty FEB list for GCs, and locations of root-inode and other areas.	0, 1, & Fixed	4.4
Bitmap	A bitmap that represents the freeness of all the FEBs.	Fixed	4.4
DirInode map	A part of the mapping table that translates a directory inode number to its physical address in the flash memory.	Fixed	4.2.1
DirInode	Attributes, file name, directory entries, and locations of the hash map.	Many	4.2.2
Hash map	A part of the hash table organized by directory entries migrated from DirInodes.	Many	4.2.2
FileInode	Attributes, file name, extent entries of file data, and locations of the extent map.	Many	4.2.3
Extent map	A set of extent entries migrated from FileInodes.	Many	4.2.3
MainData	File data indicated by extent entries.	Many	4.2.3

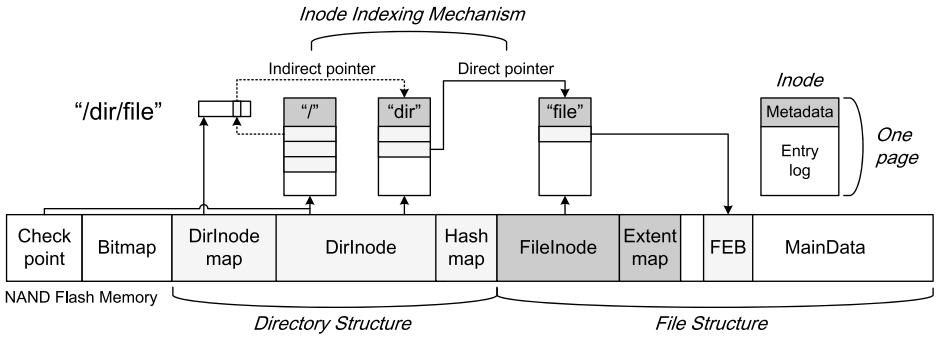


Fig. 3. Overall architecture of FlashLight.

showed that the overall GC overhead ranges from 11.93% to 27.8% over the total elapsed time (cf. Section 5), and it was mainly caused by reading obsolete pages and separating hot and cold data immaturely.

4. DESIGN AND IMPLEMENTATION OF FLASHLIGHT

4.1 Overall File System Layout

We design a novel lightweight flash file system called FlashLight based on a log-structured file system (LFS) [Rosenblum and Ousterhout 1992]. Unlike LFS that has one large log area, FlashLight maintains eight major log areas in the flash memory to maximize the effect of separating hot and cold data. Each log area occupies several FEBs and FlashLight allocates one empty FEB at a time to each log area if necessary.

Table III summarizes the name and the contents of each log area, along with the number of FEBs it requires. Note that "Many", and "Fixed" indicate multiple FEBs, and a fixed number of FEBs, respectively.

As illustrated in Figure 3, the inode occupies one page with the metadata such as its file name, inode number, file size, atime, dtime, and so on. This is possible since the page size of NAND flash memory is relatively larger than the amount of metadata [Lim and Park 2006]. This inode page is called *DirInode* or *FileInode* depending on whether the file type is a directory or a file. In addition, to handle a large number of directory entries, FlashLight creates a *hash map* page on demand. For large-sized file data, another page called *extent map* is written.

For example, when looking up a file whose pathname is "/dir/file," FlashLight performs the following procedures as illustrated in Figure 3.

(1) It obtains the root-inode by reading the page indicated by the pointer in the checkpoint data.
(2) In the root-inode page, it searches for a directory entry called "dir" and obtains its indirect pointer.
(3) It translates the indirect pointer to a physical location in the flash memory through a mapping table called *DirInode map*.
(4) It obtains the DirInode named "dir" by reading the page indicated by the translated location.
(5) In the DirInode, it finds the directory entry called "file," and finally, it obtains FileInode by reading the page indicated by the direct pointer in the directory entry. The data of "file" can be further accessed using a file data index, *extent* entry, in FileInode.

4.2 Lightweight Index Structure

Because an inode page such as DirInode or FileInode is relatively larger than the metadata size, we propose *intra-inode index logging* in which the remaining space in the inode page is used for logging indices locally. Specifically, directory entries that belong to the the same directory are logged in the parent directory's DirInode. Likewise, the information to locate its file data is kept in FileInodes. By storing both an inode and its index entries in a single page, the index lookup operation can be completed by reading one page. Similarly, when a file is created or deleted, this scheme requires only one write operation by updating the parent inode and the corresponding directory entry together. Furthermore, when an inode is updated by data write requests, it requires only one additional write operation by inserting an entry for the data and modifying the file attribute at the same time.

The following sections elaborate upon novel mechanisms for the lightweight index structure that addresses the wandering tree problem and manages the intra-inode log entries efficiently in DirInode and FileInode.

4.2.1 Inode Indexing Mechanism. Two inode indexing schemes are used in most flash file systems. One is *indirect indexing scheme* that uses a mapping table to acquire the physical location of an inode; this scheme is used in JFFS2 and YAFFS2 with in-memory structures as well as in UBIFS with a B+tree.

The other is a *direct indexing scheme* where the pointer indicates the physical page location of an inode directly. CramFS [Linux Distributor 2002], which is widely used in embedded systems as a root file system, adopts this scheme. Although the indirect indexing scheme includes the mapping table access overhead, it can easily update the physical location of an inode without any change in the parent directory entry. On the other hand, while the direct indexing scheme exhibits low access latency, the parent directory entry should also be updated whenever the location of the child inode changes, and thus, the wandering tree problem arises.

As illustrated in Figure 4(a), UBIFS adopts only indirect indexing scheme where all the physical locations of metadata and data are obtained by traversing the B+tree all the time. On the other hand, as depicted in Figure 4(b), FlashLight introduces the *hybrid indexing scheme* to address the wandering tree problem while reducing the mapping table size and enhancing the file access latency. Due to the frequent updates of inodes and the wandering tree problem, FlashLight adopts the indirect indexing scheme when pointing to DirInode. For FileInode, FlashLight employs the direct indexing scheme by substituting the inode number in the directory entry with

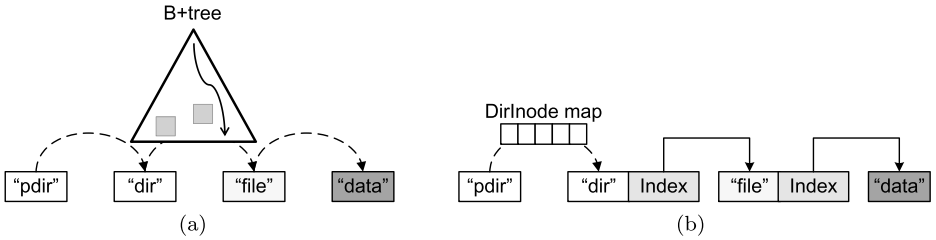


Fig. 4. Examples of the inode indexing mechanism: (a) the indirect indexing scheme in UBIFS and (b) the hybrid indexing scheme in FlashLight.

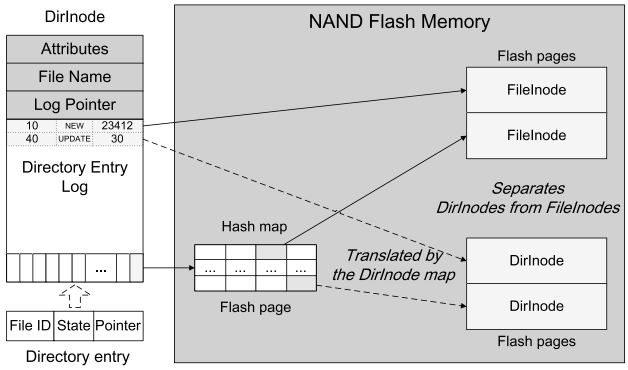


Fig. 5. Directory structure; dotted and solid lines indicate indirect and direct pointers, respectively.

the physical location of the inode. This scheme makes it possible not only to access all the child inodes directly, but also to reduce the size of the mapping table required for the indirect indexing scheme. In addition, since the physical location of a file can be used as its inode number, the file system does not have to maintain a pool of artificial inode numbers.

The mapping table for indirect index pointers is called *DirInode map*, which is organized as a linear array. It is designed for the total size not to exceed the size of one FEB. In SLC NAND chips, one 128KB-sized FEB can provide up to 32K entries. Because the number of directories in a default Linux installation is less than 20K, we believe that the size of one FEB is sufficient for storing the entire DirInode map in the embedded environment.

4.2.2 Directory Structure. As Figure 5 shows, DirInode consists of (i) directory attributes (44 bytes) that contain the basic information about the directory, such as atime, dtime, uid, gid, parent inode pointer, etc., (ii) the file name (256 bytes) that represents the name of the directory, (iii) the log pointer (4 bytes) that points to the last position of the directory entry log, (*iv*) a directory entry log area, and (v) a set of 4-byte pointers that point to 15 hash maps by default.

The directory entry log area is filled with directory entries each of which represents a child directory or a child file. Each directory entry consists of a file ID (2 bytes), a state (1 bytes), and a pointer (4 bytes). Unlike traditional file systems, a directory entry of FlashLight does not contain the full name of a file. Instead, the file name is stored in the inode page and the directory entry holds only the hashed value (file ID) of the file name. Using this file ID, child inodes that belong to the directory can be distinguished from each other. This small-sized identifier enables FlashLight to store

a large number of directory entries in one page and to look up them quickly. The state field is used to define the state of the log entry. The pointer field is assigned by the hybrid indexing scheme and indicates the location of DirInode or FileInode. If the page size is 2KB, the log area can contain approximately 240 directory entries each of which occupies 7 bytes. It means that, if users create less than 240 files in a directory, one DirInode page is enough to cover all the entries in the log area and its metadata.

In order to handle overflowed entries, FlashLight organizes a hash table. Although the B+tree works reasonably well under circumstances that numerous files are created and deleted dynamically [Litwin 1980], it may show an extra update cost, especially in flash memory, to maintain indirect index pages during split and merge operations. Instead, the hash mechanism performs an average lookup operation in *O(1)*, and it has a low update cost. FlashLight favors the simpler hash table structure because it is unnecessary to maintain a complex structure like the B+tree for a modest number of files in a directory [Agrawal et al. 2007].

The hash table is composed of a set of pages, and each page is called a *hash map*. The hash map, in turn, consists of an array of 4-byte pointers. Each hash map covers a fixed range of buckets explicitly, and to utilize the space efficiently, it is allocated on demand only when it contains more than one entry. If the size of a page is 2KB, each hash map can store 512 pointers. Because the total number of hash maps is 15 by default, the hash table can have up to 7,680 buckets in total. The buckets are contained separately in 15 hash maps; the first hash map contains #0 to #511, the second one contains #512 $\sim$ #1,023, and so on up to #7,679.

In terms of the space overhead, the B+tree used in UBIFS requires the tree space in proportion to the number of nodes. Since a node in UBIFS occupies 24 bytes for header information, UBIFS may suffer from the high space overhead. On the other hand, FlashLight occupies extra pages for the hash maps depending on the number of files in a directory and the distribution of the file IDs. If a directory runs out of the log area with a relatively small number of files having sparse file IDs, several underutilized hash maps are required. However, all the hash maps may not be required, because the directory log area contains a number of file IDs instead. Since the estimation of this space overhead can vary case by case, we compare the space overheads quantitatively in the evaluation results instead.

When a directory operation is issued, a new directory entry is appended to the directory entry log or an existing one in the log is updated. Whenever a log entry is appended or updated, its DirInode page is written to the flash memory. However, to alleviate frequent flash writes, FlashLight caches several DirInode pages in memory as described in Section 4.5. For a new entry, the file ID is calculated by

$$file\ ID = Hash(file\ name)\ \%\ (total\ \#\ of\ buckets). \tag{1}$$

The state field is set to one of NEW, UPDATE, and DELETE, depending on the current state of the file as shown in Figure 6(a). When a create request arrives, FlashLight simply appends a log entry with the NEW state. Further requests for this entry are simply processed in the log without any state change. However, once the entry is migrated to a hash map, update and delete requests are processed by appending log entries with the UPDATE and DELETE states, respectively. Subsequent updates on the log entries are absorbed in the log, and when these entries are migrated to the hash map, the corresponding hash map is finally updated.

Before migrating an entry, FlashLight checks the dedicated hash map whose number is determined by

$$hash\ map\ number = file\ ID\ /\ (\#\ of\ pointers\ in\ a\ hash\ map). \tag{2}$$

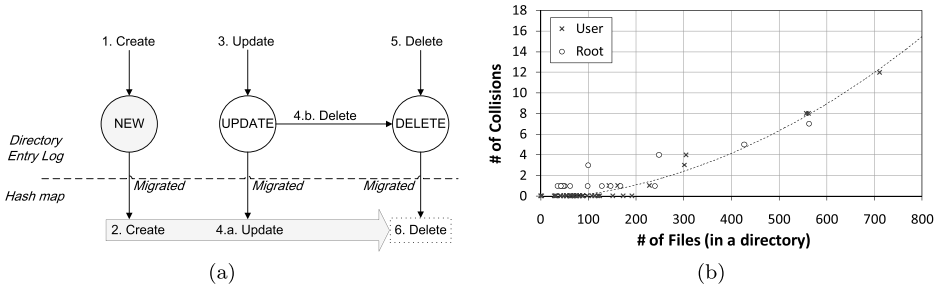


Fig. 6. (a) State transition of a directory entry log in DirInode and (b) the number of collisions in a real file set.

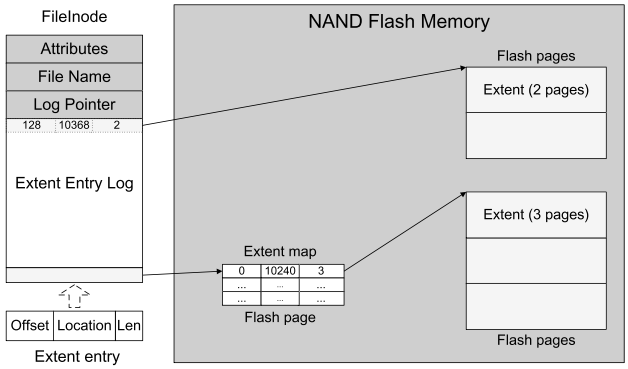


Fig. 7. File structure; only direct pointers are used.

If the hash map has already been allocated, FlashLight reads the page; otherwise, it allocates a page for the new hash map. The pointer in the victim entry is then copied to the bucket whose number is determined by

$$bucket\ number = file\ ID\ \%\ (\#\ of\ \ pointers\ in\ a\ hash\ map). \tag{3}$$

After the hash map is written to the hash map area, its physical location is recorded in the DirInode. To alleviate migrating entries frequently, all the entries that pertain to the hash map are migrated simultaneously when migrating a victim entry.

The hash collision may occur when two different file names are hashed to the same file ID. If such a collision happens, FlashLight simply keeps the collided entry in the log area without migrating it to the hash map. To estimate the probability of the hash collision in practice, we collected approximately 5,000 files made by users and 15,070 files from the default root file system in the NOKIA N810 device, and analyzed their file names. As shown in Figure 6(b), the hash collision occurred about once in every 50 files in a user directory. Since the intra-inode log can contain about 240 entries in a 2KB page, FlashLight can endure up to 240 collisions in a directory. When looking up one of the collided files in a directory, FlashLight reads the corresponding inode page until the file name matches.

4.2.3 File Structure. Figure 7 shows the structure of FileInode that is very similar to DirInode. FileInode also consists of file attributes (44 bytes), the file name (256 bytes), the log pointer (4 bytes), an extent entry log area, and locations of *extent maps*. The roles of file attributes, the file name, and the log pointer are similar to those in DirInode.

The extent entry log area is filled with *extent* entries that are similar to *nodes* in JFFS2 and UBIFS. An extent entry consists of a file offset (4 bytes), a location (4 bytes), and a length (4 bytes). The file offset indicates the data position in the file, and the location is its page offset in the flash memory. The length is the number of consecutive pages the data reside. When the page size is 2KB, the log area can contain up to 145 extent entries each of which occupies 12 bytes. In the worst case scenario where an extent entry represents 128KB of data, one FileInode page can support the file size up to 18MB. This means that a picture image generated by high-resolution digital cameras can be totally covered by only one FileInode page even under the worst case scenario.

In order to support large files, FlashLight introduces an extent map that consists of the parent directory pointer (4 bytes), its file name (256 bytes), and extent entries each having a size of 12 bytes. When the page size is 2KB, the number of extent entries in an extent map is 149. If the FileInode is filled with only extent map pointers without any log entries, it can hold 435 extent maps. In addition, if one extent has a 128KB-sized FEB, one file can cover approximately 7.9GB of data. Note that, unlike the fixed number of hash maps in DirInode, the number of extent maps dynamically changes depending on the file size.

When a new write request arrives, FlashLight writes the data to the MainData area, and then, it checks whether or not the request is followed by any existing extent. If the new data is connected to the existing data with respect to the file offset and the location, FlashLight simply extends the length of the existing extent entry; otherwise, a new extent entry is inserted into the log in FileInode. If an update request arrives, a new extent entry is also inserted to the log, and the obsolete pages are recorded and recycled by the GC mechanism.

When the extent entry log area runs out of space, some of entries in there should be migrated to the extent maps. When migrating the entries, the victim entries are selected according to their file offsets. The main policy is that each extent map has a certain range of file offsets. Once the range is determined to cover as many log entries as possible, other extent maps should contain another range of data. Through this policy, FlashLight can access the data by reading up to one extent map page additionally.

4.3 Efficient Garbage Collection Scheme

During GCs, the valid page migration overhead is a crucial performance factor. To reduce the overhead, it is necessary to achieve two goals: (i) identifying valid pages instantly, and (ii) minimizing the number of valid pages.

To identify valid pages instantly, FlashLight manages, in the checkpoint data, a list of dirty FEBs having at least one obsolete page. Each entry in the list holds its FEB number as well as a set of bits that represent the validities of all the pages in the FEB. This structure allows FlashLight to avoid unnecessary scanning of each page in the victim FEB. Since the fixed size of the checkpoint data limits the number of entries that appear on the list, FlashLight triggers GCs whenever the number of dirty FEBs exceeds a threshold. For example, when the page size is 2KB, the checkpoint data contains up to 161 dirty FEB entries. Currently, FlashLight uses a policy that GC is invoked whenever the number of dirty FEBs exceeds 140 and it reclaims 9 dirty FEBs at a time. These parameters are determined by evaluating our file system with several intensive tests. When a number of dirty FEBs are reclaimed, the system may freeze for a while, which is unacceptable in the real-time environment. To address this problem, FlashLight supports that applications can trigger GCs deliberately through a certain file system API (e.g., fstat).

To minimize the number of valid pages to move, we need to separate hot and cold data effectively. For this purpose, FlashLight separates the data into eight major log areas as mentioned in Section 4.1. This fine-grained data separation is an effective approach because each area has different hotness. For example, DirInode and FileInode are more frequently updated than the file data in the MainData area. Furthermore, FileInode is hotter than DirInode because the number of file changes caused by data writes is considerably larger than the number of directory changes.

In addition, we propose the concept of *erase-unit data allocation* in which each file has multiple FEBs, instead of pages, for its data. If an FEB is shared with several files' data, the file system may suffer from moving many valid pages, because hot and cold data can be mixed together. To mitigate the internal fragmentation that may be caused by allocating the space in a unit of FEB to a file, we propose two-level FEB lists: (i) a remnant list, and (ii) a compaction list. A remnant list holds FEBs that contain the remnants of files. Whenever a file is closed, FlashLight checks and inserts a new remnant FEB into this list. When FlashLight is almost full of data, it selects victim remnant FEBs in the Least Recently Used (LRU) order, and all the valid pages in the victims are moved to free FEBs tightly; this is called a compaction process. The compacted FEBs are finally recorded in the compaction list. In the compaction list, until all the pages in an FEB are invalidated, the FEB is kept in there without further actions. However, if FlashLight suffers from severe space pressure, more compaction processes are performed for the FEBs in the compaction list.

In reality, however, this fragmentation problem rarely occurs. We collected a number of files in use from several real users and analyzed the file patterns. Three file types, namely, movie, MP3, and picture, were collected, and the percentages over the total data size were 51.2%, 30.5%, and 18.3%, respectively. To fit the file set to our evaluation environment, we scaled down the total size by reducing the number of files proportionally, and then, we replayed many file transactions to vary the total file system utilization. The results showed that the peak total size of all the remnants occupied approximately 0.001% over the total data size, and the compaction process was triggered only when the total file system utilization was over 96.4%. Therefore, if FlashLight reserves a small amount of space for remnants, the fragmentation problem would be negligible.

4.4 Checkpoint and Bitmap Structures

Checkpoint is a well-known method for fast system boot by reading the minimum information when rebuilding the file system. When implementing this method, it is necessary to consider tracking and retrieving the checkpointed data efficiently. FlashLight adopts a two-level indirect tree that consumes four FEBs including FEB #0, FEB #1, and two other floating FEBs. FEBs #0 and #1 are root-level indirect FEBs that are used one at a time, and the others are second-level indirect and leaf FEBs, respectively. Periodically, the checkpoint data is sequentially written to the leaf FEB.

To allocate free FEBs efficiently, we propose a novel bitmap structure where each bit represents the freeness of the corresponding FEB. The i-th bit is set to zero after the i-th FEB is allocated to the major log areas, and the bit is set to one after the i-th FEB is erased. Whenever the bitmap information is changed, a new bitmap page is written to the bitmap log area. For the recovery routine, previously used FEBs should not be recycled until the file system is checkpointed newly. Initially, FlashLight reserves six FEBs by default[4] for the bitmap area to avoid frequent checkpoint operation due to the change in the bitmap information.

[4]The number of FEBs for the bitmap log can be set differently according to the file system size; six is our default number for 256MB of data capacity.

Table IV. System Environment for Experiments

System Specification	OneNAND
Platform: NOKIA N810	Part Number: KFG4G16Q2M
CPU: TI OMAP 2420	Block Size: 128KB
Memory: DDR RAM 128MB	Page Size: (2,048 + 64) Bytes
Flash Memory : OneNAND	Read Bandwidth: 108MB/s
OS: Linux 2.6.21	Write Bandwidth: 9.3MB/s
MTD: onenand.c	Erase Bandwidth: 64MB/s

4.5 In-Memory Data Structures

FlashLight keeps the following data structures in the main memory.

— The checkpoint data. The checkpoint data occupies a page, which is cached in the FlashLight's superblock information. Among the cached checkpoint data, the list of dirty FEBs is essential for selecting victims and identifying valid pages quickly during the GC process.
— A bitmap page. One bitmap page resides in the main memory all the time for FlashLight to allocate a free FEB instantly. This can cover up to 2GB or up to 160GB of storage capacity using 2KB or 4KB page, respectively.
— A DirInode map page. The DirInode map occupies at most one FEB. One page from the DirInode map is cached to retrieve frequently used mapping entries, which can cover 512 entries.
— DirInode pages. FlashLight caches two DirInode pages by default. Frequent updates on directory entries in a DirInode page can lead to a situation where FlashLight writes the DirInode page to the flash memory repeatedly. This cache can absorb those flash writes effectively.
— FileInode pages. FlashLight caches two FileInode pages by default. This cache alleviates repeated flash writes of the same FileInode page due to frequent updates of data entries.

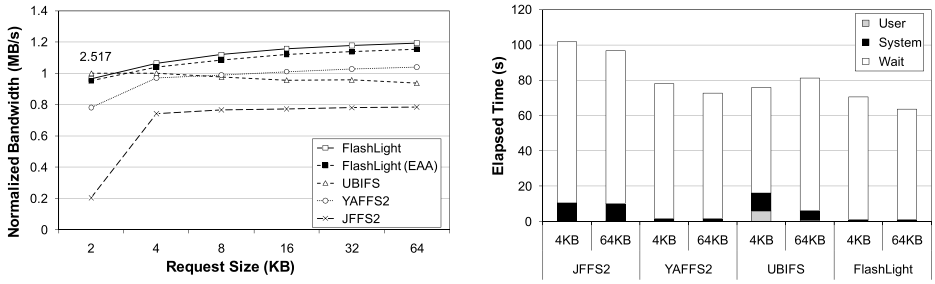
To summarize, the total amount of the main memory consumed by FlashLight is 14KB more or less when the page size is 2KB. For comparison, JFFS2 and YAFFS2 consume the memory in proportion to the number of files, and UBIFS consumes over 300KB including the space for buffering data.

5. PERFORMANCE EVALUATION

5.1 Evaluation Environment

In this section, we evaluate the performance of FlashLight. We implemented FlashLight on the Linux kernel version 2.6.21, and used NOKIA N810 as the experimental platform. The NOKIA N810 is an Internet tablet appliance, which allows the user to browse the Internet and communicate using Wi-Fi networks or with a mobile phone via Bluetooth [Nokia 2008]. In addition to the Internet activities, the N810 supports abundant multimedia services such as movie players, MP3 players, and camera functionalities. It is an embedded Linux system with a 400Mhz TI OMAP 2420 processor, 128MB DDR RAM, and 256MB OneNAND [Samsung Electronics] chip. The system parameters used in our experiments are summarized in Table IV.

We compared FlashLight with JFFS2, YAFFS2, and UBIFS. To ensure fair comparisons, we erased the entire flash space before initiating each experiment for UBIFS to avoid wear-leveling processes. In addition, we set the "no compression mode" for file data in JFFS2 and UBIFS. Because the *OneNAND* driver is not fully compatible with YAFFS2, we set *in-band tags* as a mount option to make YAFFS2 operate without the



(a) (b)

Fig. 8. Results of the SysBench benchmark: (a) normalized sequential write bandwidth and (b) elapsed time breakdown.

use of spare areas. In FlashLight, the SHA1 algorithm is used for the hash function [FIPS 180-1 1995].

Under this environment, we used three benchmark programs: SysBench [Kopytov 2004], Postmark [Katcher 1997], and Filebench [McDougall et al. 2006]. For comparison purposes, we tried to normalize the throughput of all the tested file systems as much as possible, and the absolute value was displayed over the graph as well.

SysBench was designed for evaluating a variety of components in a system running a database under intensive load. Although this benchmark supports several configurations for evaluating a system, we used the file I/O performance configuration, which supports sequential and random read/write traces. In general files created by users, such as images, MP3, and movies, most operations are requested sequentially [Evans and Kuenning 2002]. Moreover, since the read operation does not cause unnecessary read/write operations, we selected a sequential write trace to evaluate the basic performance of our file system.

Postmark is one of the most popular benchmarks, which models intensive Internet electronic mail server operations. It measures the overall transaction rate in operations/sec (ops/s) while creating and deleting numerous files in a number of subdirectories.

Filebench is a framework emulating file system workloads for evaluating system performance quickly and easily with various script files. In this benchmark, we made three workloads: *CreateFiles*, *CopyFiles*, and *Create/DeleteFiles*. The CreateFiles and CopyFiles workloads are micro-benchmarks for evaluating the basic file system performance, and the Create/DeleteFiles workload is a macro-benchmark based on a modified file server script for the user environment. Unlike the Postmark benchmark, the structure of a directory tree and the size of the file data are generated by means of a gamma distribution.

5.2 SysBench

In this experiment, one file is created, and then 190MB of data are sequentially written to the file by a single thread. Figure 8 shows the write bandwidth and the breakdown of the total elapsed time. The x-axis represents the unit of each write request. The total number of read, write, and erase operations conducted during the test are summarized in Table V. In this table, the number in parentheses denotes the erase count that is not reflected in the throughput, because UBIFS and JFFS2 erase FEBs during the test whereas FlashLight and YAFFS2 erase them in advance when the test file is deleted before the test. The 1,500 erase operations have a negligible effect on the total throughput in which they can theoretically degrade the performance by approximately 0.08MB/s. To assure this gap, we made another version of FlashLight, namely,

Table V. The Total Number of Flash I/Os in the SysBench Benchmark

Type	2KB				4KB ∼ 64KB			
	FlashLight	UBIFS	YAFFS2	JFFS2	FlashLight	UBIFS	YAFFS2	JFFS2
Read	1	31	36,468	224,859	1	31	1	49,365
Write	98,805	104,782	110,112	199,748	98,805	104,794	98,047	99,455
Erase	24(1,520)	1,550	0(1,720)	3,146	24(1,520)	1,550	1(1,720)	1,542

"FlashLight (EAA: Erase At Allocation)", which erases the obsolete FEBs during the test instead of doing so in advance. As shown in Figure 8(a), the results confirm that the performance of FlashLight (EAA) is only marginally degraded over the original FlashLight.

As shown in Table V, if the request size is a multiple of 4KB, the number of I/Os in each file system is the same, because Linux VFS (Virtual File System) issues read/write operations to the file system in the unit of 4KB. In FlashLight and UBIFS, the 2KB result is similar to the case of 4KB. This is because FlashLight processes a data request in the minimum unit of 2KB without any additional write, and UBIFS converts 2KB of data into 4KB of data through the buffer cache. In YAFFS2 and JFFS2, however, the 2KB result shows more I/Os compared to the 4KB result. In YAFFS2, in-band tag operations incur a number of additional read operations. JFFS2 has even triggered some GCs, because it wrote an additional amount of data due to the node header whose size is larger than that of YAFFS2 as well as synchronous metadata updates at every data write. Therefore, the free FEBs were consumed more quickly, and eventually, GCs were invoked in JFFS2. In UBIFS, the additional writes were mainly caused by writing node headers (22.8%), managing FEBs (22.3%), and inserting the indices of file data into the B+tree (9.4%). By contrast, FlashLight has no header structure, and extent entries were efficiently appended in the intra-inode log area of the FileInode. Additional writes were only caused by writing bitmap pages followed by allocating free FEBs.

Considering only the number of I/Os, particularly in YAFFS2 and JFFS2, we can confirm that the bandwidth increases as the request size is enlarged from 2KB to 4KB in Figure 8(a). However, UBIFS exhibits higher bandwidth in the 2KB request size rather than the 4KB case. To analyze this, we measured each elapsed time by user, system, and write completion as shown in Figure 8(b). In the 4KB case, UBIFS has a relatively larger portion of the user and system time than YAFFS2 and FlashLight. This is because UBIFS only uses a buffer cache for file data. By the buffer cache, the user time is increased, since other applications could be performed between the write requests made by the benchmark program. In addition, the system time also increases because kernel instructions were performed to manage the buffer cache. Nevertheless, the total elapsed time could be reduced with less waiting time, since the processor could interleave the execution of kernel and user instructions with the write requests. Note that, if a small number of large-sized write requests are issued, UBIFS exhibits low performance due to the lack of the interleaving effect.

As shown in Figure 8(a), the performance of all the file systems, except UBIFS, is improved slightly as the request size increases. This tendency is caused by the nature of OneNAND, which has a high burst write bandwidth through 4KB-sized buffer RAMs. We collected the time intervals between consecutive write requests in FlashLight and YAFFS2 when the request size is 64KB. These two file systems are chosen because they significantly differ in the bandwidth in spite of the similar number of I/Os. The result is shown in Figure 9 in which the y-axis represents the cumulative distribution function (CDF) for the number of write operations. This figure demonstrates that FlashLight generates a larger number of burst write operations under

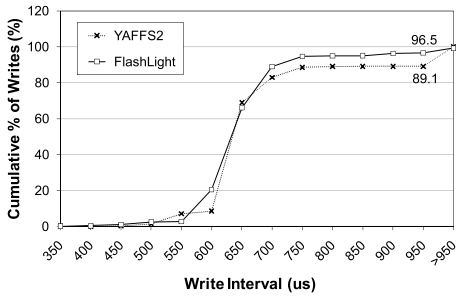


Fig. 9. Normalized CDF of number of writes according to the write intervals.

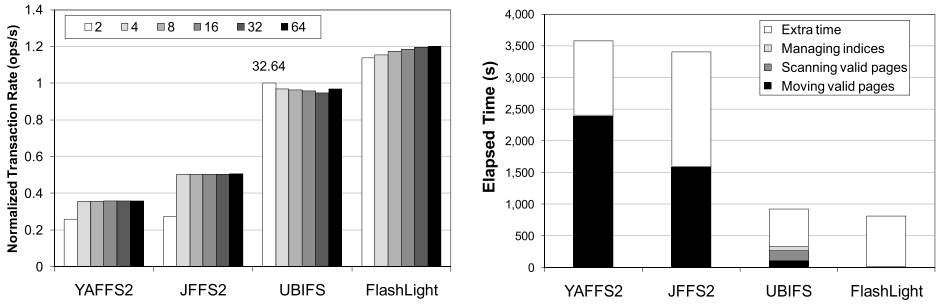


Fig. 10. Results of the Postmark benchmark: (a) normalized transaction rate according to the request size and (b) elapsed time breakdown when the request size is 2KB.

1 *ms* intervals (96.5%) than YAFFS2 (89.1%). This gap arises from the delay in in-memory operations such as copying memory-to-memory regions and handling indices in a complicated manner in YAFFS2.

Therefore, if the request size is small, the buffer cache in UBIFS helps increase the bandwidth; otherwise, the performance is affected by the nature of the OneNAND chip, rather than the buffer cache. Nevertheless, FlashLight exhibits a noticeable performance improvement across all the request sizes by mainly reducing the number of I/Os. In terms of the space overhead, UBIFS occupies about 4.7MB of flash memory space to store the index nodes of B+tree, while FlashLight stores a FileInode page (2KB) that contains file data indices, and a root-inode page (2KB) that contains the file's index. Note that, due to the sequentially written 190MB of data, the extent map is unnecessary.

5.3 Postmark

In this test, we set up a single subdirectory, and no read/append operations are performed during file system transactions. In the first CREATE phase, 1,300 files are created with 128KB of data in the root directory. In the second MIXED phase, 30,000 mixed operations consisting of create and delete operations are performed randomly; each create operation includes 128KB of data writes. Finally, in the DELETE phase, all the remaining files are deleted.

Figure 10 exhibits the performance results varying the request size of data writes. The performance is normalized to the result of UBIFS with the request size of 2KB. Since many create/delete operations are randomly performed, index management and GC costs will dominate the overall performance. To investigate the performance

Table VI. The Total Number of Flash I/Os in the Postmark Benchmark (the 2KB request size)

Type	YAFFS2	JFFS2	UBIFS	FlashLight
Read	3,091,274	12,444,884	452,129	68,464
Write	2,304,888	1,262,354	1,117,270	1,035,964
Erase	35,392	19,144	16,829	15,816

Table VII. Configurations of the Filebench Benchmark

Type	Test Name	# of Files	Directory Width	File Size	I/O Size
Micro	CreateFiles	100	100	128KB	4KB
	CopyFiles	500	20	128KB	4KB
Macro	Create/DeleteFiles	2,000	20	128KB	64KB

impact of these costs, we breakdown the elapsed time of running the Postmark benchmark as shown in Figure 10(b), and we summarize the total number of I/Os in Table VI as well.

Figure 10 confirms that the transaction rate of each file system is closely related to the index management and GC costs. YAFFS2 has a larger GC overhead than JFFS2 because YAFFS2 requires a number of write operations to handle the Tnode for data indices during GCs; nevertheless, the performance of JFFS2 is not much different to that of YAFFS2 because JFFS2 read a large amount of metadata when migrating valid nodes. The performance gap between JFFS2 and UBIFS is mainly caused by the reduced GC overhead in UBIFS; UBIFS reduces the number of valid pages to be migrated by separating FEBs between metadata and data. In addition, the use of a buffer for caching several inodes can avoid frequent writes of the updated inodes. On the contrary, as JFFS2 writes an inode at every data write, it generates a number of invalid pages, and accordingly, more GCs are triggered.

In UBIFS, 34.5% of the total elapsed time is still consumed to handle indices and GCs. Specifically, managing indices, identifying valid pages in a victim FEB, and moving them to other FEB take 6.7%, 17.1%, and 10.7% of the elapsed time, respectively. FlashLight reduces the index management overhead significantly by adopting the hybrid indexing scheme and intra-inode index logging. FlashLight decreases the GC overhead as well by the fine-grained data separation, erase-unit data allocation, and the use of a dirty list for instant identification of valid pages. The performance pattern according to the request size agrees with the SysBench results, which reflects the nature of OneNAND.

In the flash memory, UBIFS stores about 5.2MB of the tree index nodes, while FlashLight uses up to about 3.5MB of metadata including DirInode pages, FileInode pages, hash map pages, and extent map pages.

5.4 Filebench

5.4.1 Micro and Macro Tests. Table VII summarizes the configurations of each test with Filebench. Figure 11(a) and Figure 11(b) depict the resulting bandwidth and the breakdown of the elapsed time, respectively. Again, all the bandwidth results are normalized to those of UBIFS. In the Create/DeleteFiles test, the bandwidth is measured during 1,000 seconds.

For two microbenchmarks (CreateFiles and CopyFiles), Figure 11(a) shows that the performance difference between file systems is very similar to the SysBench results with the request size of 4KB. In addition, both the performance of the macrobenchmark and the breakdown results exhibit the same patterns as the Postmark

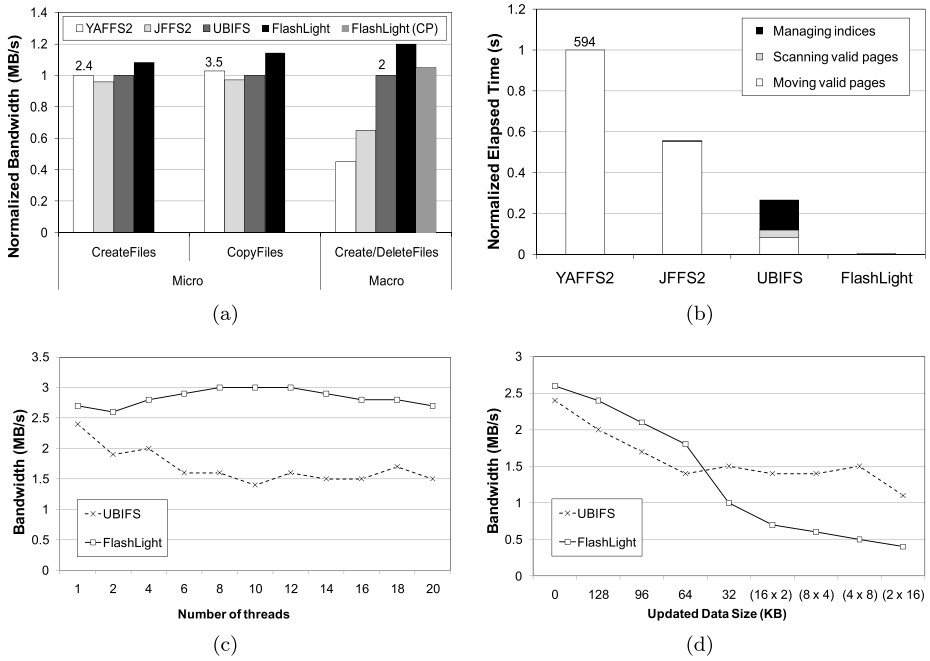


Fig. 11. Results of the Filebench benchmark: (a) normalized bandwidth, (b) normalized elapsed time in Create/DeleteFiles test, (c) real bandwidth according to the number of threads, and (d) real bandwidth according to the amount of updated data and the update patterns.

results. In the Create/DeleteFiles results, FlashLight exhibits about 20% of the performance improvement over UBIFS because it reduces the index management and GC overhead considerably. During the Create/DeleteFiles test, UBIFS occupies about 6.3MB of flash memory space to store the index nodes of B+tree, while FlashLight takes about 2.6MB for metadata.

As described in Section 4.3, the compaction process reads, writes, and erases one FEB-sized data, which causes some delays. To measure this overhead quantitatively, we made another version of FlashLight, namely, "FlashLight (CP: Compaction Process)", which triggers the compaction process very intensively whenever 50 remnants are newly created. During the Create/DeleteFiles test, FlashLight read/wrote 204,724 pages and erased 11,769 FEBs during the compaction process; the percentages over the total number of I/Os were 71.8%, 14.2%, and 44%, respectively. As shown in Figure 11(a), the overall performance is degraded by 12.5% over the original performance of FlashLight that seldom triggers the compaction process. This means that, even if the compaction process is triggered heavily, FlashLight can tolerate without degrading the performance significantly.

5.4.2 Multiple Threads Tests. Figure 11(c) compares the performance of UBIFS and FlashLight when we vary the number of concurrent threads during the test. The basic configuration is the same as in the Create/DeleteFiles test described in Table VII, and each thread performs four operations repeatedly until the time is over: (1) CREATE that creates a new file and writes 128KB of data sequentially to the file, (2) APPEND that opens a file and appends 128KB of data, (3) DELETE that selects a file randomly and deletes it, and (4) STAT that requests a file's stat.

From Figure 11(c), we can observe that FlashLight outperforms UBIFS for all the cases. Until the number of threads reaches 10, the performance of FlashLight improves gradually. This is due to that FlashLight can afford to process multiple operations. However, this advantage disappears as the number of threads becomes greater than 10. FlashLight then suffers from I/O contention. On the other hand, the performance of UBIFS is getting worse as the number of threads increases because UBIFS already has bounded capabilities.

5.4.3 File Update Tests. Recently, many camcorders support editing services; customers can cut or paste the data in video clips, and modify the pictures with variety of visual effects. To verify the effect of these services in performance, we made file update tests as follows.

The basic configuration is also the same as in the Create/DeleteFiles test described in Table VII, and a single process is triggered with four sequences: (*1*) CREATE, (*2*) RANDOM_WRITE, (*3*) DELETE, and (*4*) STAT. CREATE, DELETE, and STAT sequences are the same as in the previous multi-thread test described in Section 5.4.2. RANDOM_WRITE opens a file and randomly writes 32KB∼128KB of data to the file.

Figure 11(d) shows the performance degradation rate of UBIFS and FlashLight according to the amount of updated data and the update patterns. All the tests are categorized into two parts. One is updating between 0KB and 128KB of data in a file as one extent. The other is updating 2, 4, 8, and 16 extents while the total size of updates is fixed to 32KB. As shown in Figure 11(fig-filebench), the results of the former tests are shown in the left-hand side; those of the latter are shown in the right-hand side and they are represented as "(the extent size × the number of extents)." Note that the "(2 × 16)" test performs the most scattered data updates, leading to the worst performance.

We can see that FlashLight achieves better performance than UBIFS if the data is updated as one extent; however, if many scattered data are updated in a file, UBIFS exhibits better performance. However, this performance degradation is just due to the GC policy of FlashLight; FlashLight performs GCs more frequently than UBIFS to preserve the fixed number of FEBs in the dirty list. This means, on the other side, that FlashLight is advantageous to prepare a large volume of free space for unexpected large-sized data writes.

5.4.4 Real-Time Tests. FlashLight may freeze the system momentarily to perform GCs. To address this problem, FlashLight allows the user applications to trigger GCs through a certain file system API, as mentioned in Section 4.3. To ensure whether this can work or not, we made a real-time test as follows.

(1) For aging the FlashLight file system, we first performed the Create/DeleteFiles test shown in Section 5.4.1.
(2) To make a room to write consecutive data, we deleted several files in the test directory.
(3a) One test measured the latencies of every write requests while writing the total 64MB of data in a unit of 64KB.
(3b) Unlike the previous test, the other test invoked the API to trigger GCs before every write requests, and measured the latencies.

Figure 12 shows the latencies of data writes before and after using this API in the application. We can see that the peak latency is decreased from 512.1ms to 304.9ms by invoking GCs at user level. For the real-time environment, therefore, applications may call the API on the fly before requiring a low latency during the data writes.

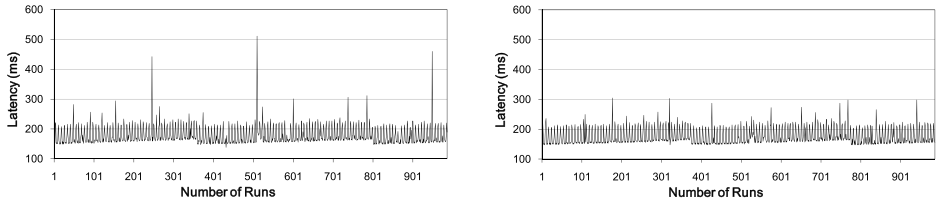


Fig. 12. Latencies of data writes: (a) before and (b) after triggering GCs deliberately.

6. CONCLUSION

In this article, we investigate two design issues for high-performance flash file system. One issue is to design an efficient index structure that locates where files and data reside in the flash memory. With the increasing capacity of embedded systems, the practical use of both JFFS2 and YAFFS2 has become difficult because they require a large amount of memory. To reduce the memory consumption, UBIFS has been developed with an on-flash index structure; however, it degrades the system performance to manage the index structure, B+tree. The other issue is to design an efficient GC scheme. During GC, identifying and moving valid pages, called valid page migration, can cause a considerable number of additional read and write operations. To identify valid pages instantly, another data structure is required, and to minimize the number of valid pages, hot and cold data must be separated effectively.

We present FlashLight, a lightweight, high-performance flash file system that has the following features: (*i*) a lightweight index structure that introduces the hybrid indexing scheme and intra-inode index logging to reduce the index management overhead and (*ii*) an efficient GC scheme that adopts the fine-grained data separation, erase-unit data allocation, and a list of dirty FEBs for instant identification of valid pages. Our experimental results confirm that FlashLight alleviates index management and GC overheads by up to 33.8% over UBIFS with the Postmark benchmark. Furthermore, we demonstrate that FlashLight improves the overall performance by up to 27.4% over UBIFS with the SysBench benchmark.

REFERENCES

AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. 2007. A five-year study of file-system metadata. *ACM Trans. Storage 3,* 3.

ALEPH ONE LTD. 2003. Yet another flash file system v2 (yaffs2). http://www.yaffs.net.

BITYUTSKIY, A. 2005. JFFS3 design issues. http://www.linux-mtd.infradead.org/.

CHANG, L.-P., KUO, T.-W., AND LO, S.-W. 2004. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst. 3,* 4, 837–863.

CHOI, H.-J., LIM, S. H., AND PARK, K. H. 2009. Jftl: A flash translation layer based on a journal remapping for flash memory. *ACM Trans. Storage 4,* 4.

DOUGLIS, F., CÁCERES, R., KAASHOEK, M., LI, K., MARSH, B., AND TAUBER, J. 1994. Storage alternatives for mobile computers. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI'94)*. 25–37.

EVANS, K. M. AND KUENNING, G. H. 2002. A study of irregularities in file-size distributions. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'02)*.

FIPS 180-1. 1995. Secure hash standard. U.S. Department of Commerce/N.I.S.T.

GAL, E. AND TOLEDO, S. 2005. A transactional flash file system for microcontrollers. In *Proceedings of the USENIX Annual Technical Conference*. 89–104.

HUNTER, A. 2008. A brief introduction to the design of ubifs. http://www.linux-mtd.infradead.org.

KATCHER, J. 1997. Postmark: A new file system benchmark. In *the TR3022 of Network Appliance*.

KIM, J., KIM, J., NOH, S., MIN, S., AND CHO, Y. 2002. A space efficient flash translation layer for compactflash systems. *IEEE Trans. Consumer Electron. 48,* 2, 366–375.

KOPYTOV, A. 2004. SysBench: A system performance benchmark. http://sysbench.sourceforge. net/index.html.

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.

LINUX DISTRIBUTOR. 2002. CRAMFS (Compressed ROM file system). http://lxr.linux.no/source/fs/cramfs/README.

LITWIN, W. 1980. Linear hashing: A new tool for file and table addressing. In *Proceedings of 6th International Conference on Very Large Data Bases (VLDB'80)*. 212–223.

MCDOUGALL, R., CRASE, J., AND DEBNATH, S. 2006. FileBench: File system microbenchmarks. http://www.opensolaris.org.

NOKIA. 2008. N810 Internet tablet. http://www.nokiausa.com/A4626058.

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

SAMSUNG ELECTRONICS. www.samsung.com/global/business/semiconductor/.

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.

WOODHOUSE, D. 2001. Jffs: The journalling flash file system. In *Proceedings of the Ottawa Linux Symposium*.

Received June 2009; revised January 2010; accepted April 2010