JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory

HYUN JIN CHOI, SEUNG-HO LIM, and KYU HO PARK
Korea Advanced Institute of Science and Technology

In flash memory-based storage, a Flash Translation Layer (FTL) manages the mapping between the logical addresses of a file system and the physical addresses of the flash memory. When a journaling file system is set up on the FTL, the consistency of the file system is guaranteed by duplications of the same file system changes in both the journal region of the file system and the home locations of the changes. However, these duplications inevitably degrade the performance of the file system. In this article we present an efficient FTL, called *JFTL*, based on a journal remapping technique. The FTL uses an address mapping method to write all the data to a new region in a process known as an out-of-place update. Because of this process, the existing data in flash memory is not overwritten by such an update. By using this characteristic of the FTL, the JFTL remaps addresses of the logged file system changes to addresses of the home locations of the changes, instead of writing the changes once more to flash memory. Thus, the JFTL efficiently eliminates redundant data in the flash memory as well as preserving the consistency of the journaling file system. Our experiments confirm that, when associated with a writeback or ordered mode of a conventional EXT3 file system, the JFTL enhances the performance of EXT3 by up to 20%. Furthermore, when the JFTL operates with a journaled mode of EXT3, there is almost a twofold performance gain in many cases. Moreover, the recovery performance of the JFTL is much better than that of the FTL.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; D.4.3 [**Operating Systems**]: File Systems Management—*Maintenance*; B.3.2 [**Memory Structures**]: Design Styles—*Mass storage (e.g., magnetic, optical, RAID)*

General Terms: Design, Performance, Algorithms

Additional Key Words and Phrases: Flash memory, flash translation layer, journaling file system, journal remapping, garbage detection

ACM Reference Format:

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, Article 14 (January 2009), 22 pages. DOI = 10.1145/1480439.1480443 http://doi.acm.org/10.1145/1480439.1480443

Authors' addresses: H. J. Choi, S.-H. Lim, and K. H. Park, Korea Advanced Institute of Science and Technology, 335 Gwahangno (373-1 Guseong-dong), Yuseong-gu, Daejeon 305-701, Republic of Korea; email: {hjchoi, shlim}@core.kaist.ac.kr; kpark@ee.kaist.ac.kr.

1. INTRODUCTION

With a dramatic increase in capacity during the last few years, flash memory has become one of the major components of data storage. As flash memory becomes a viable alternative to disks, it is accordingly becoming an interesting field of study that demands the attention of researchers. The deployment of flash memory has spread rapidly in applications ranging from consumer electronic devices to general purpose computer architecture, due largely to its useful features of nonvolatility, small size, shock resistance, and low power consumption [Douglis and Tauber 1994].

In spite of the advantages of flash memory, many flash memory operations are hampered by physical restrictions. In-place updates are not possible in flash memory, which means a programmed byte must be erased before it can be changed. In addition, a large block of flash memory must be erased whenever a programmed byte is erased. Furthermore, the erasing count of each block is limited and the erase operation itself hinders the system performance. The write operation is consequently more important than the read operation in flash memory-based systems. There are two different approaches to solve these problems. One approach involves the design of a native file system that considers specific characteristics of flash memory, such as JFFS [Woodhouse 2001], YAFFS [Aleph One Ltd. 2007], and CFFS [Lim and Park 2006]. The other approach involves the introduction of a software layer called a Flash Translation Layer (FTL), which is placed between traditional file systems and flash memory [Intel Corporation 2009]. Our focus is on the FTL. The FTL enables flash memory to fully emulate the functionality of a normal block device, mainly through its unique logical to physical address mapping, garbage collection, and wear leveling techniques. The advantage of this emulation is that any traditional file system, such as the FAT file system [Microsoft 2000], can be directly set in flash memory via the FTL.

A journaling technique is essential for the availability, reliability, and consistency of a file system [Chutani and Sidebotham 1992]. The journaling file system obviates the need for a time-consuming consistency check of the entire file system when recovering from a failure. For the recovery, the file system only checks the auxiliary log, which contains the most recent file system changes. Once the journaling file system is constructed on the FTL, the consistency of the file system is guaranteed, with the duplicated copies of the same file system changes as those written in the log and in the home locations of the changes. However, these duplications degrade the system performance.

A number of commercial robust FAT file systems use the FTL. The TFAT file system of Microsoft ensures that the file allocation table remains intact by managing two file allocation tables, where one can replace the other if the system crashes. Samsung Electronics has developed a commercial FAT-based journaling file system. TFS [Samsung Electronics 2009b] and RFS [Samsung Electronics 2009a] both use journaling features to enhance the robustness of the FAT file system. Park and Ohm developed an atomic write operation for the robustness of the FAT file system [Park and Ohm 2005], though this approach has some drawbacks in that it cannot protect directory

entries and its performance is limited because the atomic unit is a single write operation.

We present an FTL equipped with an efficient journal remapping technique. The proposed FTL, called JFTL, eliminates redundant duplicates by remapping addresses of journaled (logged) blocks of the file system changes to addresses of real file system blocks. Because of the characteristic of out-of-place updates of flash memory, the journal remapping technique of the JFTL places little additional overhead on the system. The JFTL also preserves the consistency of file systems.

The remainder of this article is organized as follows. Section 2 describes the relevant background and motivation for this work. Section 3 describes the design and implementation of the proposed technique, and Section 4 presents the experimental results. We present our concluding remarks in Section 5.

2. BACKGROUND AND MOTIVATION

2.1 Characteristics of Flash Memories

Flash memory has two types of nonvolatile memory: NOR and NAND. A NOR flash memory has a faster random access speed; however, it has a higher cost and lower density than a NAND flash memory. In contrast to NOR flash memory, NAND flash memory has the advantages of a large storage capacity and relatively high performance for large read and write requests. Our proposed JFTL can be best understood in terms of NAND flash memory, though it can be associated with both types of flash memory.

Flash memory is arranged into blocks where each block has a fixed number of pages. A page is further divided into a data region for storing data and a spare region for storing the status of the data region, such as an error correction code. The read or write access to data is on a per-page basis in NAND flash memory. The erase is done on a per-block basis. Flash memory has three physical constraints. First, in-place updates of data are not permitted because all the written areas of flash memory should be erased before they can be reprogrammed. Thus, out-of-place updates are essential for the design of flash file systems. Second, the erase unit of a block is significantly larger than the program unit of a page. In a recent flash chip, the size of a block is 128kB, whereas that of a page is 2kB. Because of this size difference, a cleaning operation such as garbage collection must move live pages from an obsolete block into a free block before erasing the obsolete block. Third, the life-span of flash memory is limited by the number of erase cycles. Generally, each flash block is guaranteed a maximum of 100,000 erase cycles.

2.2 Flash Translation Layer

The FTL overcomes with these constraints of flash memory by emulating flash memory as a block device. The FTL was originally patented by Ban [1995], and numerous FTL techniques have been proposed in the literature, including address mapping, garbage collection, and wear leveling [Gal and Toledo

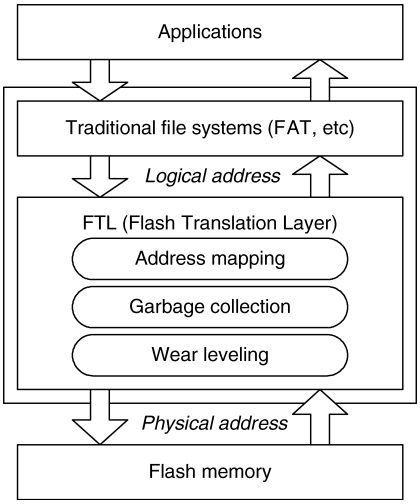


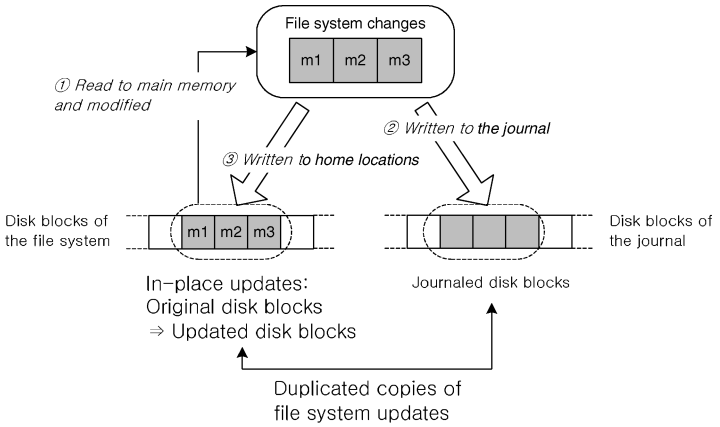
Fig. 1. System architecture of a flash memory system that uses the FTL.

2005a, 2005b; Chang and Kuo 2004; Chang 2007]. Figure 1 shows the system architecture of a flash memory system that uses the FTL.

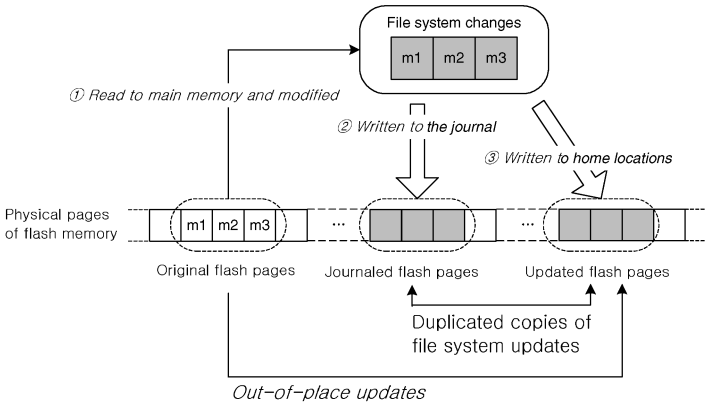
In the FTL, each write request of a dirty block of a file system is redirected to an empty physical page of flash memory that has been erased in advance. The FTL keeps track of the address mapping information between the logical blocks of a file system and the physical pages of flash memory. In this manner, the FTL prevents in-place updates of data and hides the latency of the erase operation. When free spaces are reclaimed in the flash memory, the FTL performs the garbage collection: It finds partially obsolete flash blocks, moves valid data from the obsolete blocks to free blocks, and then erases the obsolete blocks. The FTL also uses a wear leveling technique to wear down flash blocks as evenly as possible for a long life-span of flash memory.

2.3 Journaling File System

When unpredictable system failures occur, a file system goes into an inconsistent state. For system availability, the file system must be returned quickly to a consistent state. Of the numerous proposals that address this issue, journaling techniques are the most promising and have been applied to current popular file systems such as EXT3 [Ts'o and Tweedie 2002], ReiserFS [Reiser 2007], XFS [Sweeney and Peck 1996], and JFS [Best and Haddad 2003]. In these journaling file systems, the important file system changes are recoded in a special part of a storage called a journal, before being written back to their home locations in the file system. This process obviates, in the file system recovery, the need for a time-consuming consistency check of the entire file system, which is performed in nonjournaled file systems. Instead, a little of the pending changes of the file system, recorded in the journal, are checked for the recover and replayed. Thus, it generally takes only a few seconds for the journaling file system to recover from a failure.



(a) when constructed on a disk



(b) when constructed on flash memory via the FTL

Fig. 2. Duplicated copies of file system changes in a journaling file system.

2.4 Research Motivation

While the journaling technique guarantees the consistency of the file system, it has a considerable overhead, as shown in Figure 2; more specifically, the same file system changes in the main memory are written to storage twice: once to the journal and once to their home locations in the file system. When the journaling file system is constructed on a disk, in-place updates of the file system changes are performed. In other words, as shown in Figure 2(a), after the changes are written to the journal, they overwrite disk blocks containing the original version of data. On the other hand, when a journaling file system is set up on flash memory via the FTL, the aspect of the duplicated copies is different. As shown in Figure 2(b), after being journaled, the file system changes are written out of place in the original flash pages by virtue of the address mapping of the FTL. Thus, once the changes are committed to flash memory, both the original flash pages and the duplicated copies of the changes (the journaled and updated flash pages in the figure) exist in the flash memory. However, the write operation of

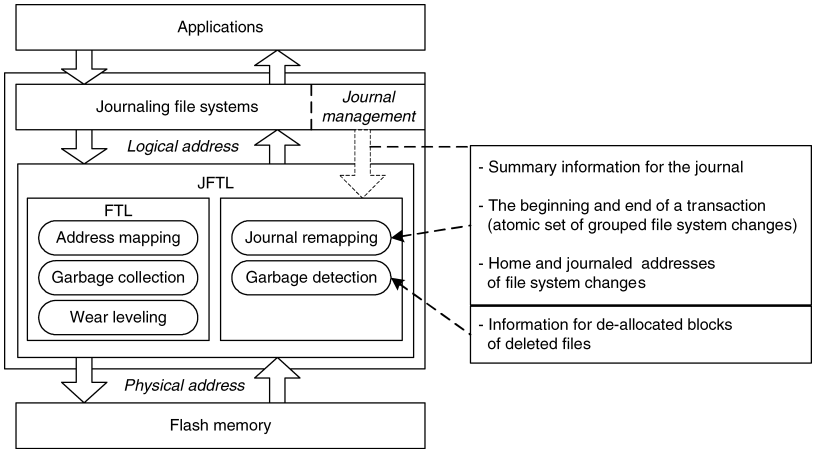


Fig. 3. System architecture of a flash memory system that uses the JFTL.

the file system changes to their home locations (the second write operation in the figure) is redundant in the sense that the journaling file system can recover to be a consistent state only with the original and journaled flash pages; that is, without the updated flash pages. This redundant write operation involves a considerable overhead and consequently degrades the performance of the file system; it also causes more garbage collection and rapid wear of flash memory.

As mentioned, the duplication is a crucial feature for the consistency of a file system; moreover, the FTL performs out-of-place data updates by means of its address mapping management. The problem when the FTL is associated with the journaling file system is that the FTL has no awareness of the journaling operations. Specifically, it lacks knowledge of what data is duplicated, and of where the home and journaled locations of the data are. The management and processing of this information in the FTL would eliminate redundant and unnecessary writings of file system changes to flash memory. The JFTL is proposed for this purpose.

3. JFTL: AN FTL BASED ON JOURNAL REMAPPING

In this section we provide an overview of the proposed JFTL, along with details of its design and implementation.

3.1 Overview

Figure 3 shows the system architecture of a flash memory system that uses the JFTL. The JFTL is associated with journaling file systems and emulates flash memory as a block device. The JFTL has unique techniques of journal remapping and garbage detection, as well as FTL techniques of address mapping, garbage collection, and wear leveling. All the necessary information for managing the journal of the file systems are delivered to the JFTL, including summary information for the journal, the beginning and end of a transaction consisting of an atomic set of grouped file system changes, and home and journaled addresses of file system changes.

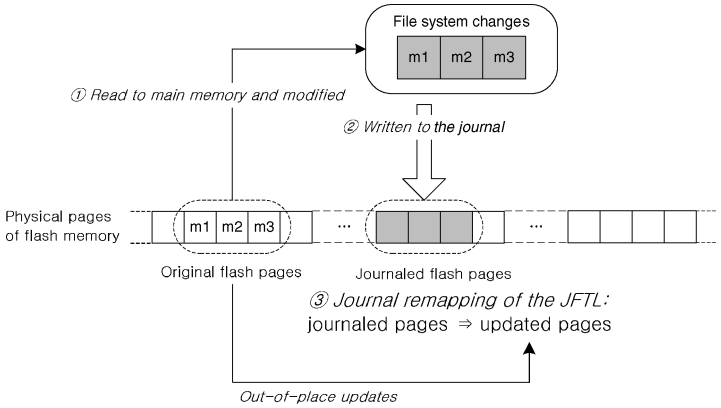


Fig. 4. Journal remapping technique of the JFTL.

3.1.1 *Journal Remapping Technique.* Figure 4 illustrates the journal remapping technique of the JFTL. After writing file system changes to the journal in flash memory, the JFTL does not write the changes again to the storage. Instead, with the information from the journaling file system in Figure 3, the JFTL remaps the addresses of the journaled pages to addresses of home locations of the changes. With this technique, the journaled pages are altered into updated pages of the changes; thus, redundant writings to flash memory are prohibited.

3.1.2 *Garbage Detection Technique.* In the conventional disk-based file system, the delete operation of a file does not erase data blocks of the file but only modifies the related metadata, from which deallocated data blocks become free blocks. In the flash-memory-based file system, however, the characteristic of the out-of-place updates of the FTL prohibits deallocated flash pages from being used for data writing until they are reallocated by the file system. This is because the FTL lacks knowledge of what flash pages are deallocated, and thus considers those deallocated pages as valid pages. The garbage collector of the FTL might perform unnecessary moving of the deallocated pages to free flash blocks, though they actually comprise garbage no longer in use.

As shown in Figure 5, this problem is solved by integrating the garbage detection technique into the JFTL. As shown in Figure 3, the information of deallocated blocks of deleted files is delivered to the JFTL. By using this information, the JFTL detects the corresponding deallocated flash pages and marks them as obsolete in advance. With this technique, the garbage collector of the JFTL is more efficient and runs less often than that of the FTL.

3.2 Design and Implementation

In designing and implementing the JFTL, we chose the EXT3 file system from among the numerous kinds of journaling file systems, primarily because of its easy accessibility. To implement the JFTL, we used a Memory Technology Device (MTD) driver in Linux [MTD 2009].

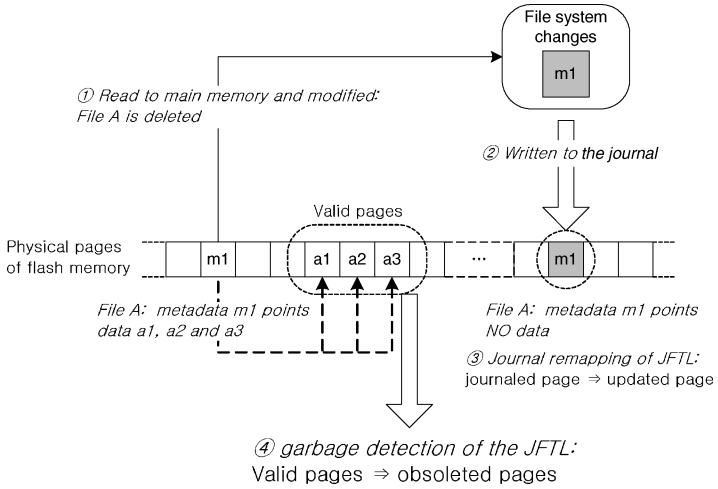


Fig. 5. Garbage detection technique of the JFTL.

3.2.1 *Briefs of the EXT3 File System.* EXT3 aims to maintain complex file system structures in a consistent state. All the important file system changes are first written to the journal. After the writes of the journal are safely done, the normal write operations occur in the home locations of the changes. The journal region is recycled after the normal updates are done. A fundamental unit of the EXT3 file system is a *transaction* used for the consistency of the file system. Whenever a system call modifies the file system, dirty data and metadata blocks (file system changes) resulting from the system call are grouped in the transaction and handled in an atomic way. When a system has recovered from a failure, EXT3 ensures that either all the grouped blocks in the transaction are applied to the file system or none of them is applied. The descriptor of the transaction has two major lists for grouping data and metadata blocks: The *t_sync_datalist* list is a doubly linked circular list of data blocks to be directly written to the home locations of the blocks; the *t_buffers* list is a doubly linked circular list of dirty blocks to be journaled.

The transaction exists in several states over its lifetime, namely, the *running*, *commit*, and *checkpoint* states. In the running state, as shown in Figure 6(a), the dirty data and metadata blocks are grouped into the two transaction lists. In some bounded conditions, such as a transaction interval timeout or a synchronization request, the running transaction is closed and goes to the commit state. At this time, a new transaction is created to gather subsequent file system changes. In the commit state, the closed transaction performs the task of writing its data and metadata blocks to storage. As shown in Figure 6(b), all the data blocks of the t_sync_datalist list are committed to their home locations, and the metadata blocks of the t_buffers list are then committed to the journal. The task of writing to the journal begins with a special block called a Journal Descriptor (JD) block, which contains the information of the fixed home locations of subsequent journaled blocks. At the end of the commit, another special block called a Journal Commit (JC) block is attached to identify the end

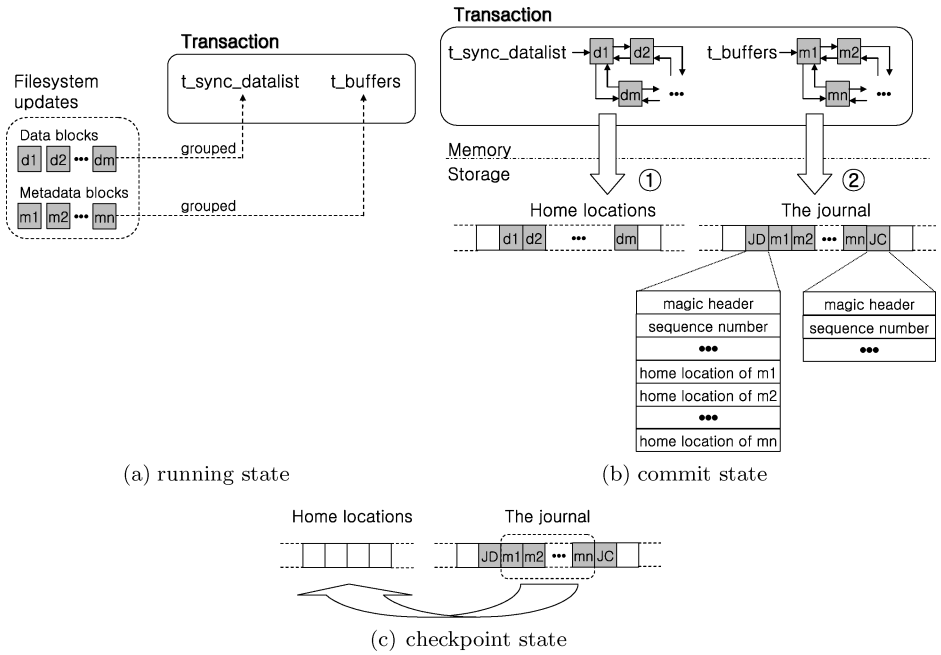
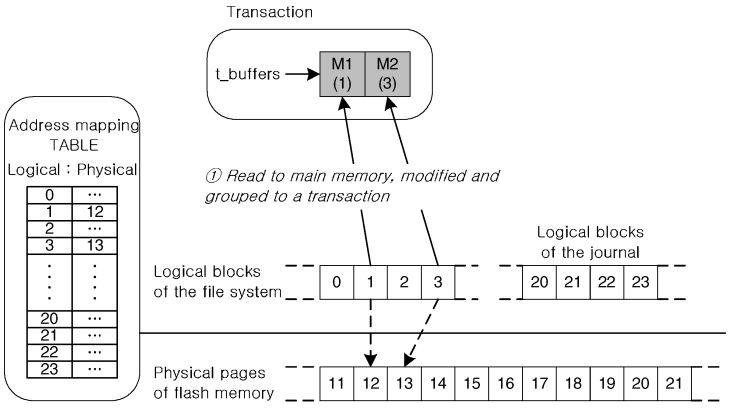


Fig. 6. States of a transaction in EXT3.

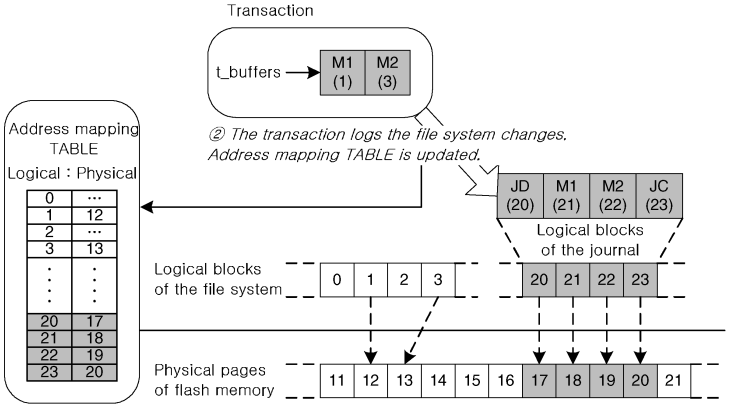
of the transaction. Each transaction can be distinguished with the sequence number written in the JD and JC blocks. In the checkpoint state, as shown in Figure 6(c), both the journaled data blocks (in the case of the journaled mode) and the metadata blocks of the transaction are copied to their home locations in storage. When the checkpointing is completed, the transaction is removed from the journal region and its space is reclaimed. If a crash occurs before or during the checkpoint of a committed transaction, the transaction is checkpointed again during the file system recovery.

3.2.2 *Journal Remapping and Garbage Detection.* When a journaling file system is built on the JFTL, the file system changes that have been journaled to flash memory are not actually rewritten to the medium, but are remapped. The journaling information is delivered through the dedicated interface between the journal management module of EXT3, called a journal block device, and the JFTL. With this information, the JFTL performs address remapping of journaled blocks.

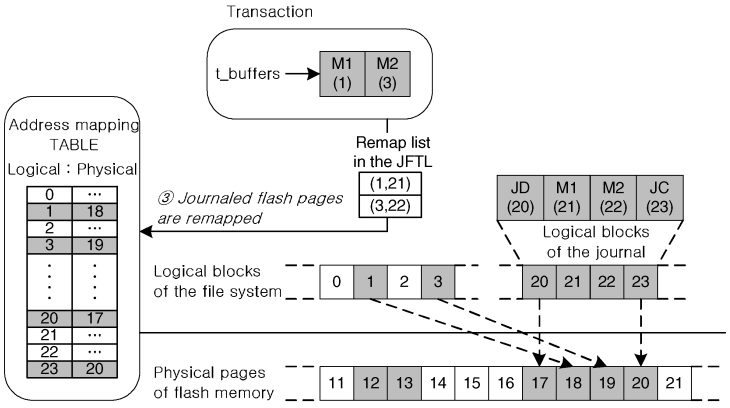
The overall process of the journal remapping is described as follows with the example presented in Figure 7. As shown in Figure 7(a), after a new transaction is started, the file system changes whose home addresses (or logical block numbers) are 1 and 3 are grouped to the transaction and then linked and locked. In some bounded conditions, the transaction enters the commit state. During this state, as shown in Figure 7(b), the grouped file system changes of the transaction are written to the journal with two additional special blocks, the JD and JC blocks, which have a magic header and a sequence number identifying



(a) file system changes are grouped during the transaction running state



(b) file system changes are journaled during the transaction commit state



(c) addresses of journaled flash pages are remapped to their home addresses

Fig. 7. The overall process of journal remapping of the JFTL.

the commit transaction. The home addresses 1 and 3 of subsequent journaled blocks are recorded to the JD block in the order of the blocks being written. The JFTL writes the journaled blocks to new empty flash pages associated with the journal and then updates its address mapping table. After the journal flushing, the journal block device module of EXT3 sends the journaling information of the JD block to the JFTL for the journal remapping. By using the information, as shown in Figure 7(c), the JFTL makes a remap list whose elements (1, 21) and (3, 22) comprise pairs of logical home addresses and logical journaled addresses of the file system changes. With the remap list, the JFTL remaps the physical addresses of journaled flash pages 18 and 19 to their logical home addresses 1 and 3, respectively, within its address mapping table. Once this remapping process for the journaled pages has been completed, the file system changes should not be written again to flash memory because the logical home addresses of the changes are now mapped to the physical addresses of the newly updated flash pages. Accordingly, the file system changes in the main memory are cleaned by a simple process of clearing the *dirty* flags of the memory buffers associated with the changes. As a result, the changes are not flushed to flash memory by the kernel flushing daemon *pdflush*. After the completion of all these processes, the JFTL renders flash pages 12 and 13 obsolete; these pages were previously mapped to logical home addresses 1 and 3. These obsolete pages are marked as dead and included in the erase operation of flash memory. The transaction is then removed.

The garbage detection of the JFTL is implemented as follows: In EXT3, the information of deallocated blocks is traced in the *b_committed_data* field of the journal head data structure [Bovet and Cesati 2005]. This field references dynamically allocated memory associated with a metadata block that contains block bitmap information. Whenever data blocks of a file are deallocated, the information of the blocks is recorded in the memory referenced by the field. During the transaction commit state, the set of *b_committed_data* fields containing the information of the deallocated blocks are delivered to the JFTL. With the information, the JFTL detects deallocated flash pages, which are garbage, and renders them obsolete in advance.

3.3 Recovery

A recovery utility of the operating system associated with the JFTL scans the EXT3 journal to return the file system to a consistent state. The recovery processing proceeds as follows: The recovery utility identifies transactions in the journal by inspecting sequence numbers of the transaction. For each complete, or properly committed transaction, the recovery utility reads the JD blocks of the transaction, and extracts and sends the journal information to the JFTL. Just like when run-time transactions are committed, after receiving the information, the JFTL makes a remap list for the journaled blocks and performs the journal remapping within its address mapping table. The recovery utility repeatedly conducts this process until it finds an incomplete or improperly committed transaction in the journal. At the end of the recovery, the utility resets the journal of EXT3 to be used later for journaling operations.

In contrast with the disk-based recovery of file systems, the JFTL does not copy the journaled blocks of complete transactions, but instead remaps them to the home locations of the blocks. Thus, even though a few blocks are involved in the file system recovery, this recovery mechanism is more efficient than the previous one, which needs copies of the journaled blocks to their homes.

3.4 Issues of Data Consistency and Performance

The journaling file system supports the three journaling modes, namely, the *writeback*, *ordered*, and *journaled* modes. In the writeback mode, only metadata blocks are logged to the journal. Data blocks are written directly to their home locations of the file system. This mode does not enforce any writing order between data blocks and metadata blocks. Thus, the metadata blocks can be written to storage before the data blocks are written and vice versa, which accordingly provides no guarantee of data consistency if the system fails, but affords the highest runtime performance of the three journaling modes. In the ordered mode, only metadata blocks are logged. However, unlike the writeback mode, the ordered mode ensures that data blocks are written to their home locations before the related metadata blocks are written to the journal. This writing order prevents the inconsistency problem of the writeback mode. In other words, unwritten data blocks are never referenced by any metadata block. In the journaled mode, full-data journaling is carried out. Data and metadata blocks which have relations are written together to the journal and later propagated to their home locations. The journaled mode provides the best guarantee of data consistency, though it generally has a much slower runtime performance than the other modes due to the penalty of full-data journaling.

When the three journaling modes are associated with the JFTL, their characteristics differ slightly from those observed in the disk-based storage. The writeback and ordered modes yield more overhead to the flash memory-based storage, which is mainly due to the out-of-place data updates of the JFTL: In the recovery phase of EXT3, incompletely written data blocks which are not referenced by any metadata are considered as free blocks by the file system semantics. However, the JFTL regards these blocks as valid blocks because they have been already used for data writing. The JFTL cannot use these blocks for data writing until they are reallocated by the file system, which may impose garbage collection overheads.

On the other hand, the journaled mode removes the burden of its major overhead, namely, full-data journaling, through the journal remapping technique of the JFTL. The data is merely remapped to home locations instead of being duplicated.

3.5 Issues of Delivering Journal Information

A solid state disk was developed with a multiple array of flash chips for use as a storage that could replace or be used in conjunction with hard disks [Weiler 2007]. In the same manner as disks, the solid state disk is generally connected by the ATA or Serial ATA (SATA) interface of the main board of a system. In this configuration, the main barrier to adopting the JFTL is a requirement that the

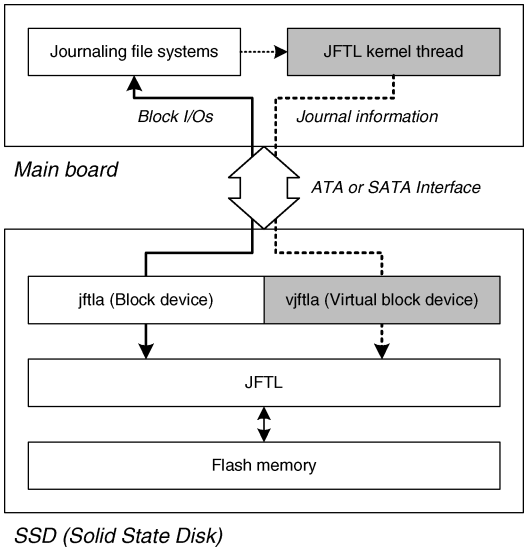


Fig. 8. Dedicated interface for the delivery of journal information.

journal information must be delivered through the ATA or SATA interface to the JFTL of the solid state disk. One solution for delivering the journal information is making a dedicated interface for the delivery, as shown in Figure 8, by means of a virtual block device of the solid state disk. A JFTL kernel thread associated with journaling file systems extracts all the necessary information of the journal, and delivers the information to the JFTL via the dedicated interface. The JFTL then can use the delivered information for the journal remapping.

4. EVALUATION

The experimental environment includes an Intel Pentium 4 PC system with 1.6 GHz of CPU, and 512MB of RAM. The kernel version is Linux-2.6.9. Of the many types of FTL drivers, the FTL driver in the MTD package of Linux is used for our experiments [MTD 2009]. The FTL driver was modified to support NAND flash memory and the proposed JFTL was then implemented on the modified driver. The address mapping technique used in the drivers is sector mapping, where every logical sector (or block) is mapped to a corresponding physical flash page. To emulate a NAND flash device, we used the *nandsim* simulator included in the MTD package. The nandsim can be configured with various NAND flash devices in accordance with the associated physical characteristics. The physical characteristics of the simulated flash memory are as follows: The configured flash memory has a page size of 512 bytes and a block size of 16kB; each page has a read time of 25 us, and a programming time of 200 us; and each block has an erase time of 2 ms. These configured settings are accepted in many types of flash memory. The capacity of the simulated NAND flash memory was set to 256MB, which was allocated in the main memory of the system, resulting in 256MB of available system memory.

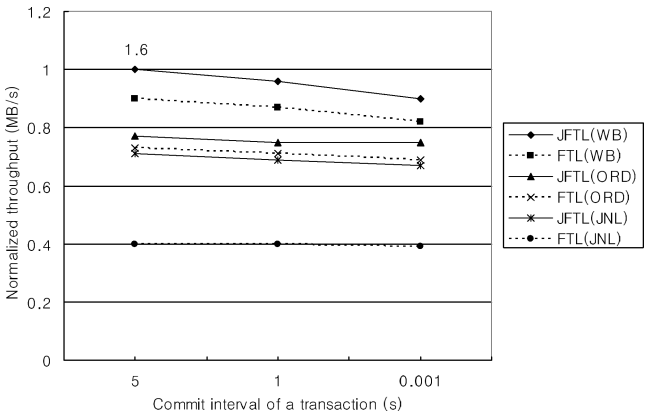


Fig. 9. The results of the Tar benchmark.

We compared the JFTL to the FTL with three journaling modes of EXT3 in Linux. The JFTL versions operating with the writeback, ordered, and journaled modes of EXT3 are denoted as JFTL(WB), JFTL(ORD), and JFTL(JNL), respectively. In the same manner, the FTL versions are denoted as FTL(WB), FTL(ORD), and FTL(JNL). The block size of EXT3 was set to 1kB and the journal size was set to 4MB.

We conducted two sets of benchmark tests to evaluate the performance of the proposed JFTL. One set contains file system benchmarks which exercise the metadata operations of tested file systems, such as *mkdir*, *rmdir*, *create*, *unlink*, and *readdir*. The other set contains file input/output (I/O) benchmarks, which measure the data transfer rates of tested file systems. All the benchmarks were run with a cold file system cache for which the test file system was remounted whenever a test was performed.

4.1 File System Benchmarks

To evaluate the file system performance, we used four benchmark programs: *Tar* command in Linux, *Postmark* [Katcher 1997], *Dbench* [Vieira and Madeira 2003], and *Filebench* [McDougall and Debnath 2006] benchmarks. For comparison purposes, we normalized the throughput of all the tested file systems with respect to the throughput of the JFTL(WB). The absolute throughput was written over each JFTL(WB) point or bar of the benchmark results.

4.1.1 *Tar.* The Tar benchmark, which we made, extracts files from an archive that stored the *include* directory of the kernel source tree. The archive consisted of 1668 subdirectories and 7117 files with a mean file size of 3kB and a standard deviation of 23kB.

Figure 9 shows the results of the benchmark. The throughput of each file system is plotted as the commit interval of an EXT3 transaction; and the commit interval was 5 s, 1 s, and 0.001 s. The Tar benchmark performs numerous metadata operations that manipulate the data structures of flash memory; for instance, by locating a free block in the file system, updating the data block bitmap

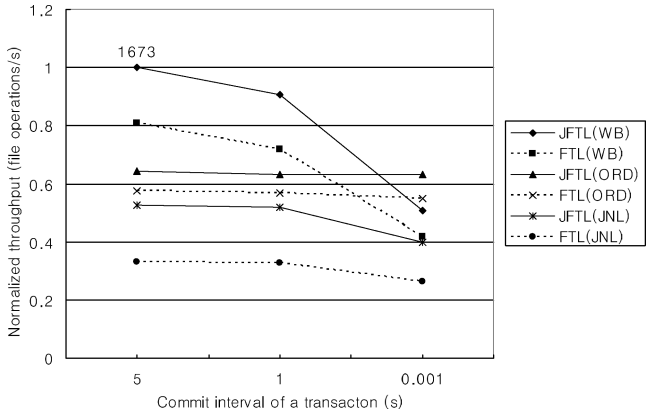


Fig. 10. The results of the Postmark benchmark.

inside the proper block group, and updating several fields of *inodes*. During the Tar benchmark, 9781 metadata blocks were updated and 81,693 data blocks were updated. Because of the absence of duplication of metadata and data (in the case of the JFTL(JNL)), the JFTL performs better than the FTL. Compared with the FTL, the JFTL is about 10% faster in the writeback mode and 5% faster in the ordered mode; moreover, the performance gap between the JFTL and the FTL is maintained, regardless of the commit interval of an EXT3 transaction. In the journaled mode, the JFTL outperforms the FTL by roughly 70%.

4.1.2 *Postmark.* Postmark was designed to measure the transaction rates for a workload approximating a large Internet electronic mail server [Katcher 1997]. The benchmark (version 1.5) initially creates 15,000 files with sizes ranging from 512 bytes to 4kB; the files are spread across 500 subdirectories. The benchmark performs 15,000 file operations, including the *create*, *delete*, *append*, and *read* file operations. There is no bias in selecting any particular file operation.

Figure 10 shows the results of the benchmark. As with the Tar benchmark, the Postmark benchmark performs numerous metadata operations pertaining to the operations of small files. Thus, the throughput results are similar to those of the Tar benchmark; that is, the JFTL is faster than the FTL by about 20% in the writeback mode, 10% in the ordered mode, and 60% in the journaled modes. In contrast to the Tar benchmark, when the commit interval was set to 0.001 s, the ordered mode outperformed the writeback mode. This difference in performance is due to the fact that the real commit interval of the ordered mode was much longer than that of the writeback mode because the implementation of EXT3 involved the use of a piggy-backing technique. Note, however, that the performance gap between the JFTL and the FTL is maintained, as in the case of the Tar benchmark.

4.1.3 *Dbench.* Dbench was designed to simulate a disk I/O load of a system running the NetBench benchmark suite [Memik and Hu 2001], which is used to rate Windows file servers. Dbench produces only a file system load and performs

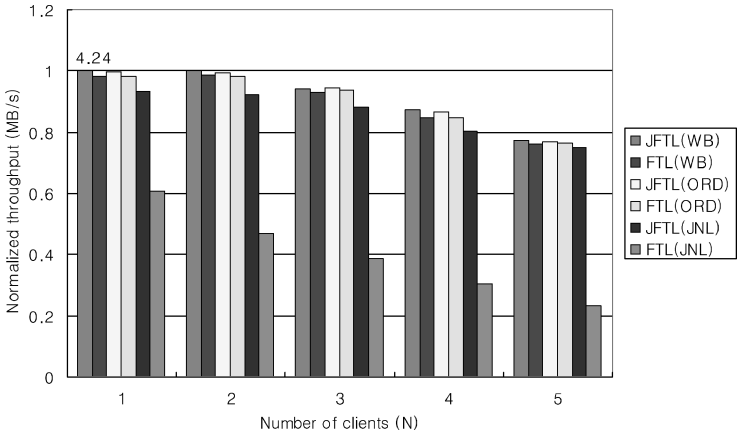


Fig. 11. The results of the Dbench benchmark.

Table I. Configurations of Filebench Benchmark

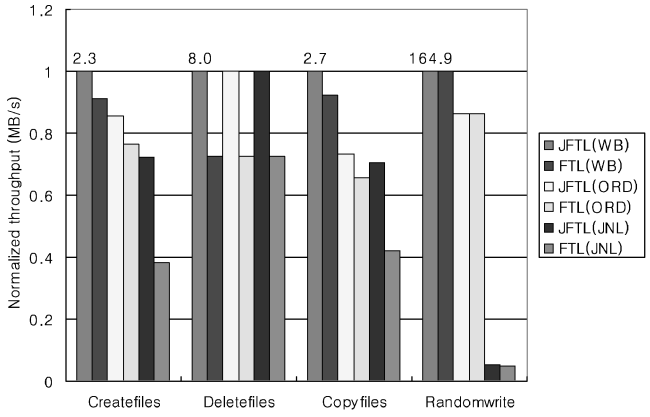
	Number of Files	Directory Width	File Size	Number of Threads	I/O Size
Createfiles	50,000	50	2kB	16	(n/a)
Deletefiles	50,000	50	2kB	16	(n/a)
Copyfiles	25,000	25	2kB	1	(n/a)
Randomwrite	1	1	1MB	1	8 kB
Varmail	1,000	1,000,000	16kB	16	16 kB
Mongo	25,000	50	2kB	1	2 kB
Webserver	1,000	20	16kB	50	16 kB

all of the same I/O calls that an smbd server in Samba produces when confronted with the Netbench run. The test output is the true throughput as seen by the benchmark. We executed this benchmark for 1 to 5 simulated clients. The commit interval of an EXT3 transaction was set to 5 s, and the runtime of the benchmark was 90 s.

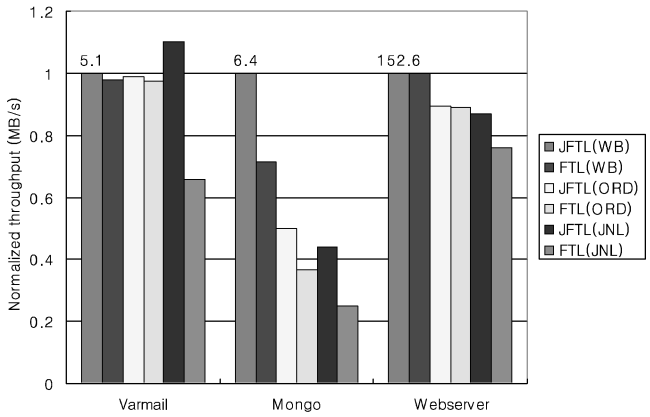
Figure 11 shows the benchmark results in which the throughput results of file systems are plotted as the number of simulated clients increases. Dbench performs file operations mainly for 1MB of files; thus, the number of metadata updates in this benchmark is much smaller than that of the previous benchmarks. The JFTL consequently performs slightly better than the FTL in the writeback and ordered modes. In contrast, the JFTL is vastly superior to the FTL in the journaled mode by virtue of the journal remapping of data blocks.

4.1.4 *Filebench.* FileBench, which is a framework of file system workloads for measuring and comparing file system performance, was developed as part of a strategy for the performance characterization of local and NFS file systems. Table I summarizes the configurations of each subtest of Filebench. The commit interval of an EXT3 transaction was set to 5 s.

Figure 12 shows the results of the benchmark. From the results of all subtests, we confirm that the JFTL is always better than the FTL, regardless of the



(a) microbenchmarks



(b) macrobenchmarks

Fig. 12. The results of the Filebench benchmark.

kind of journaling mode being used. In the writeback or ordered mode, the JFTL performs over 10% better than the FTL for file system workloads producing lots of metadata updates, such as *createfiles*, *deletefiles*, or *mongo* tests. On the other hand, the JFTL performs similarly to the FTL for file system workloads producing little metadata updates, such as *randomwrite* or *webserver* tests. In the journaled mode, the JFTL outperforms the FTL, delivering more than a 50% performance gain in most sub-benchmarks.

4.2 File I/O Benchmarks

To evaluate the file I/O performance, we used three benchmark programs: Bonnie [Bray 2009], Tiobench [Kuoppala 2004], and IOzone [Norcott 2009] benchmarks. While running the benchmarks, we guaranteed that all the data would be written to flash memory without being cached to the main memory.

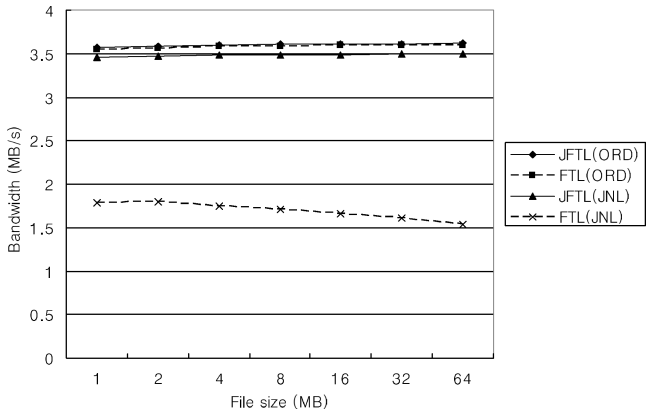


Fig. 13. The results of the Bonnie benchmark. The graph plots the block write performance as a file increases in size along the x-axis.

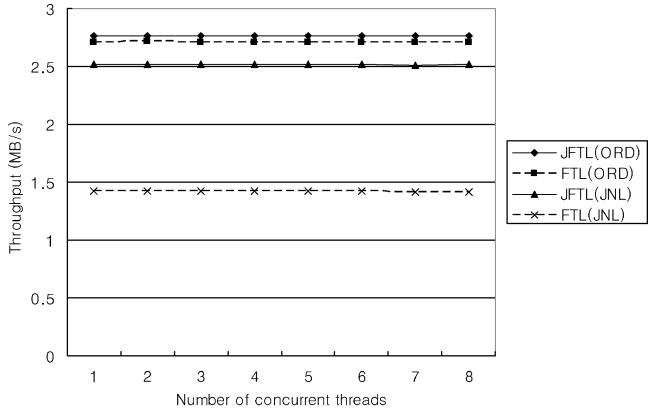
We excluded the results of the JFTL(WB) and the FTL(WB) because they performed very similarly to the JFTL(ORD) and the FTL(ORD), respectively.

4.2.1 *Bonnie.* Bonnie performs a series of tests for a single file of known size, where the tests are composed of character write/read, block write/read/rewrite, and random write. We chose to evaluate the block write performance as the file increased in size. In block writes, a file is written by the *write*() system call with a request size of 4kB.

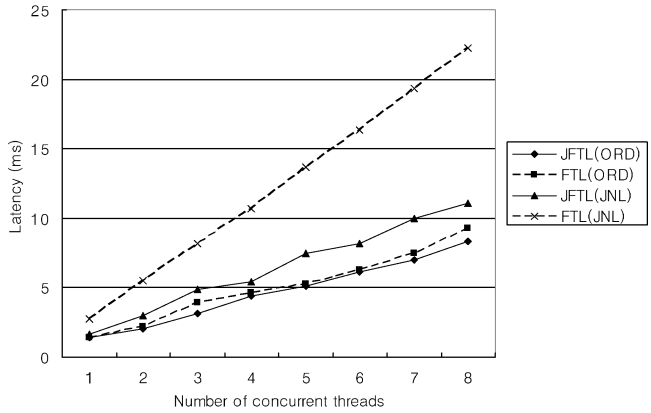
Figure 13 shows the results of the benchmark. We can see that the bandwidth of each file system is not substantially changed as the file size increases. In the graph, the JFTL(ORD) slightly outperforms the FTL(ORD) because of the benefit of the journal remapping of the metadata. However, the performance gap is quite small because the amount of metadata updates in this file I/O benchmark is negligible compared to the amount of data updates. When the JFTL(JNL) is compared with two ordered file systems, namely, the JFTL(ORD) and the FTL(ORD), the JFTL(JNL) performs similarly to ordered file systems as well as providing a higher level of data consistency. When comparing the two journaled file systems, we found as expected that the JFTL(JNL) greatly outperforms the FTL(JNL) because the former conducts journal remapping of both data and metadata. For the results of the FTL(JNL), the bandwidth is slowly decreased as the file increases in size; this outcome is a result of the size limit of the journal configured to 4MB.

4.2.2 *Tiobench.* Tiobench is a benchmark especially designed to test the I/O performance of a file system with multiple running threads. We evaluated the random write performance while increasing the number of concurrent threads, each of which randomly selects a 4kB block of a test file and writes it to flash memory. The number of random writes of each thread was set to 500.

Figure 14 shows the throughput and latency of the random writes where the latency means the average time taken to write one block of 4kB to the test file. The trend of the throughput results is similar to that of the Bonnie benchmark.



(a) throughput of random writes



(b) latency of random writes

Fig. 14. The results of the Tiobench benchmark. The graph plots the throughput and latency as the number of concurrent threads increases along the x-axis.

The results show that flash memory is not affected by the randomness of requests, which are due to a distinct feature of flash memory: Specifically, as an electrical device, flash memory is not affected by any geometric structure. Flash memory is free from a complex scheduling overhead such as disk scheduling. When we examine the latency results, we can see that the JFTL outperforms the FTL, and the average latency of each file system increases in proportion to the number of concurrent threads. As the result, the latency gap between file systems gets enlarged.

4.2.3 *IOzone.* IOzone generates and measures a variety of file operations, including write/rewrite, read/reread, and aio read/write. Of these, we measured the write performance of a file whose size would become 4096kB at the end of the benchmark, while we varied the request size of write operations from 4kB to 4096kB.

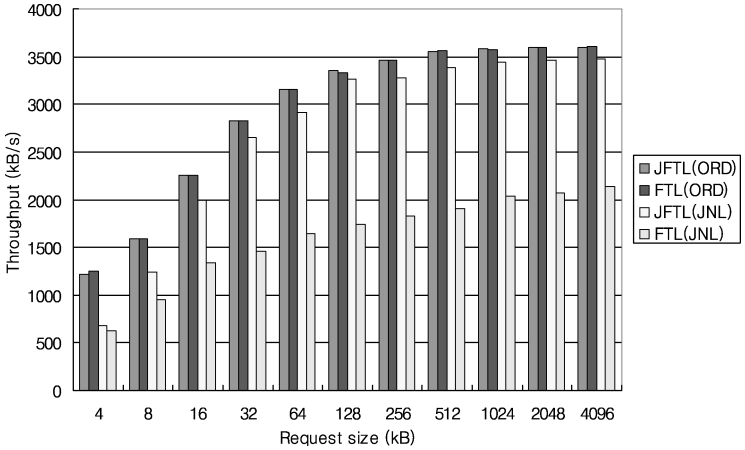


Fig. 15. The results of the IOzone benchmark. The graph plots the write performance as a request size of the write operation increases along the x-axis.

Table II. Mount Time of EXT3 with a 4MB Journal

	Reconstruction of Address Mapping Table	Recovery of File System	Mount Time
FTL	400 ms	0–1,800 ms	400–2,200 ms
JFTL	400 ms	0–32 ms	400–432 ms

Figure 15 shows the results of the benchmark. We can see that more write throughput can be obtained with an increase in the request size of write operations. This outcome occurs from the overhead in writing journal special blocks, such as journal descriptors, to the journal. The increase in the request size reduces the total number of write operations of the benchmark, and thus lowers the total number of journal special blocks to be written, which heightens the throughput. In the graph, the JFTL(ORD) performs similarly to the FTL(ORD) for the same reason as with the previous file I/O benchmarks. On the other hand, the performance gap between JFTL(JNL) and FTL(JNL) is enlarged as the request size increases because the overhead in writing journal special blocks is reduced, whereas the overhead of data logging remains unchanged.

4.3 Mount Time of File Systems

The mount time of EXT3 is mainly determined by the reconstruction of address mapping table from flash memory to the main memory, and the recovery of the file system. Table II shows the mount time of EXT3 with a 4MB journal. To accomplish the reconstruction, the FTL or JFTL reads the spare regions of the flash memory that contain the address mapping information and makes a mapping table from this information; this reconstruction process takes 400 ms with our simulated flash memory configuration. For the recovery, the FTL reads the journaled blocks of complete transactions in the journal, and copies the blocks to their home locations. Thus, the recovery time of the FTL depends largely on the journal size of EXT3 and on the number of blocks that are copied

during the recovery. With our simulated configuration, this recovery process of the FTL takes between 0 ms and 1800 ms. On the other hand, the JFTL reads the descriptor blocks of complete transactions and remaps journaled blocks to their home locations. In this recovery process of the JFTL, only the spare regions of flash pages are used for the writing of the remapped addresses. Thus, the JFTL provides a faster recovery of EXT3 than the FTL.

5. CONCLUSION

In the last few years, flash memory has become one of the major components of data storage because of its dramatic increase in capacity. Nonetheless, in spite of its many advantages for data storage, flash memory is hampered in its operations by its physical characteristics. The major drawbacks of flash memory are, firstly, that bits can only be cleared by erasing a large block of memory and, secondly, the erasing count of each block has a limited number. In the case of a journaling file system on an FTL, duplications of file system changes between the journal and the home location of the changes are crucial for the reliability and consistency of the file system. However, these duplications degrade the file system performance.

We present an efficient journaling interface between a file system and flash memory. The proposed FTL, called JFTL, eliminates redundant duplications by remapping journaled locations of file system changes to the home locations of the changes. Thus, the JFTL prevents the file system constructed on it from being degraded by the duplications, as well as preserving the consistency of the file system. Our approach is a layered approach, which means that any conventional file system can be set up with our method with no or minimal modification. Our benchmark evaluations reveal several important results. First, when the writeback or ordered journaling mode is used, the JFTL performs better than the FTL owing to the help of the journal remapping. The JFTL also provides the same level of data consistency as the FTL. With file system workloads accompanying massive metadata operations, the JFTL can outperform the FTL by more than 10%. On the other hand, with file I/O workloads producing little metadata updates, the JFTL only slightly outperforms the FTL. Second, when the journaled mode is used, the FTL seriously degrades the system performance due to the logging of both data and metadata changes; in many cases, the throughput and latency are half that of the writeback or ordered mode. In contrast, the JFTL does not degrade the system performance severely and provides a substantially smaller overhead to the system than the FTL. Third, the JFTL enables faster recovery of a file system than the FTL because the former does not copy, but remaps, the journaled blocks to their home locations as long as the file system recovers from a failure.

REFERENCES

SWEENEY, A., DOUCETTE, D., W. H. C. A. M. N., AND PECK, G. 1996. Scalability in the xfs file system. In *Proceedings of the USENIX Annual Technical Conference.*

ALEPH ONE LTD. 2007. Yaffs: A nand-flash filesystem. http://www.aleph1.co.uk/yaffs.

BAN, A. 1995. Flash file system. U.S. Patent 5404485.

BOVET, D. P. AND CESATI, M. 2005. *Understanding the Linux Kernel,* 3rd ed. O'Reilly & Associates.

BRAY, T. 2009. Bonnie benchmark. http://textuality.com/bonnie.
CHANG, L.-P. 2007. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the ACM Symposium on Applied Computing*.
CHANG, L.-P. AND KUO, T.-W. 2004. An efficient management scheme for large-scale flash-memory storage systems. In *Proceedings of the ACM Symposium on Applied Computing*.
DOUGLIS, F., CACERES, R., F. K. K. L. B. M., AND TAUBER, J. A. 1994. Storage alternatives for mobile computers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, 25–37.
G. MEMIK, W. H. M.-S. AND HU, W. 2001. Netbench: A benchmarking suite for network processors. In *Proceedings of the International Conference on Computer Aided Design*.
GAL, E. AND TOLEDO, S. 2005a. Algorithms and data structures for flash memories. *ACM Comput. Surv. 37,* 2.
GAL, E. AND TOLEDO, S. 2005b. Mapping structures for flash memories: Techniques and open problems. In *Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering*.
INTEL CORPORATION. 2009. Understanding the flash translation layer(ftl) specification. http:// developer. intel.com.
KATCHER, J. 1997. Postmark: A new file system benchmark. Tech. rep.
KUOPPALA, M. 2004. Threaded I/O bench for Linux. http://directory.fsf.org/tiobench.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. 2009. Memory technology device (mtd) subsystem for linux. http://www.linux-mtd.infradead.org.
MICROSOFT. 2000. Fat: General overview of on-disk format.
NORCOTT, W. D. Ioznoe filesystem benchmark. http://www.iozone.org.
R. MCDOUGALL, J. C. AND DEBNATH, S. 2006. Filebench: File system microbenchmarks. http://www.opensolaris.org/os/community/performance/filebench.
REISER, H. 2007. Reiserfs. www.namesys.com.
BEST, S., D. G., AND HADDAD, I. 2003. IBM's journaled filesystem. *Linux J*.
S. CHUTANI, O. ANDERSON, M. K. B. L. W. M. AND SIDEBOTHAM, R. 1992. The episode file system. In *Proceedings of the Winter USENIX Conference.*
S. PARK, J. Y. AND OHM, S. 2005. Atomic write FTL for robust flash file system. In *Proceedings of the 9th International Symposium on Consumer Electronics*.
SAMSUNG ELECTRONICS. 2009a. RFS Samsung's flash software solution optimized to use nand flash memory on linux based ce device.
SAMSUNG ELECTRONICS. 2009b. TFS4 Samsung's integrated flash software solution for nand flash memory.
TS'O, T. AND TWEEDIE, S. 2002. Future directions for the ext2/3 filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*.
VIEIRA, M. AND MADEIRA, H. 2003. A dependability benchmark for OLTP application environments. In *Proceedings of the International Conference on Very Large Data Bases*.
WEILER, Y. 2007. White paper: Bringing solid state drive benefits to computer notebook users. http://www.sandisk.com/OEM/WhitePapers.
WOODHOUSE, D. 2001. Jffs: The journalling flash file system. In *Proceedings of the Ottawa Linux Symposium*.

Received September 2007; revised December 2008; accepted December 2008