Journal Remap-Based FTL for Journaling File System with Flash Memory

Seung-Ho Lim, Hyun Jin Choi, and Kyu Ho Park

Computer Engineering Research Laboratory,
Department of Electrical Engineering and Computer Science,
Korea Advanced Institute of Science and Technology,
Daejon, South Korea
{shlim,hjchoi}@core.kaist.ac.kr, kpark@ee.kaist.ac.kr

Abstract. Constructing flash memory based storage, FTL (Flash Translation Layer) manages mapping between logical address and physical address. Since FTL writes every data to new region by its mapping method, the previous data is not overwritten by new write operation. When a journaling file system is set up upon FTL, it duplicates data between the journal region and its home location for the file system consistency. However, the duplication degrades the performance. In this paper, we present an efficient journal remap-based FTL. The proposed FTL, called $JFTL$, eliminates the redundant data duplication by remapping the journal region data path to home location of file system. Thus, our $JFTL$ can prevent from degrading write performance of file system while preserving file system consistency.

1 Introduction

Flash Memory has become one of the major components of data storage since its capacity has been increased dramatically during last few years. It is characterized by non-volatile, small size, shock resistance, and low power consumption [1]. The deployment of flash memory is spreading out ranging from consumer electronic devices to general purpose computer architecture. In nonvolatile memories, NOR flash memory provides a fast random access speed, but it has high cost and low density compared with NAND flash memory. In contrast to NOR flash memory, NAND flash memory has advantages of a large storage capacity and relatively high performance for large read/write requests. Recently, the capacity of a NAND flash memory chip became gigabyte stage, and the size will be increased quickly. NAND flash memory chips are arranged into blocks, and each block has a fixed number of pages. Currently, typical page size of NAND is 2 KB, and the block size is 128 KB.

For flash memory, many operations are hampered by its physical characteristics. First, the most important feature, is that bits can only be cleared by erasing a large block of flash memory, which means that in-place updates are not allowed. It is called out-of-place update characteristics. When data are modified, the new

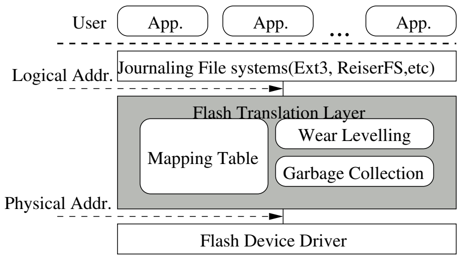


Fig. 1. Layered Approach with FTL

data must be written to an available page in another position, and old page becomes dead page. As time passes, a large portion of flash memory is composed of dead pages, and the system should reclaim the dead pages for writing operations. The erase operation makes dead pages become available again. Second, erasing count of each block has a limited number and erase operation itself gives much overhead to system performance. Thus, the write operation is more important than read operation when designing flash memory based systems.

Due to its limitations, data robustness and consistency is a crucial problem. Since every data should be written to new place in flash memory, a sudden power failure of system crash may do harm the file system consistency. There are two approaches to address these problems. One approach would be possible by designing a file system itself to support specific flash memory characteristics. There are several native flash file system, such as JFFS [8] and YAFFS [9]. These are log-structured based file system that sequentially stores the nodes containing data and metadata in every free region, although their specific structures and strategies are different from each other. These approach are available only on board flash memory chips, and if it is desired to be used in PC systems, a flash memory interfaces should be integrated into computer architecture.

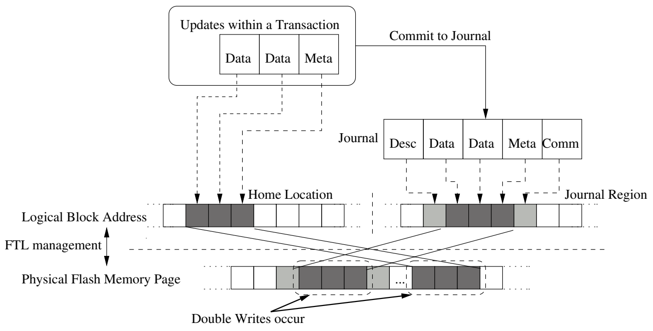
The other approach is to introduce a special layer, called a Flash Translation Layer (FTL), between existing file system and flash memory [4][5][6]. The FTL remaps the location of the updated data from one page to another and manages the current physical location of each data in the mapping table. FTL gives a block device driver interface to file system by emulating a hard disk drive with flash memory through its unique mapping algorithm. Figure 1 shows the layered approach with FTL in flash memory based systems. This layered approach can offload flash memory from processing unit by utilizing commonly used interfaces such as SCSI or ATA between processor and flash memory chips. The removable flash memory cards such as USB mass storage and compact flash are using this approach. The advantage of it is that existing file systems can be used directly on the FTL. When existing file systems use upon a FTL, journaling technique is essential element to prevent from corrupting user data and enhance system reliability, data consistency and robust. However, journaling method existing file system itself degrades the system performance. Generally, journaling file systems write a journal record of important file system data before processing write operation. Then this journal is re-written to real file system home location to make clear consistent state. Due to out-of-place update characteristics of flash memory, the journaling affects flash memory based system more than disk based system.

In this paper, we present an efficient journaling interface between file system and flash memory. The proposed FTL, called $JFTL$, eliminates the redundant data copy by remapping the logical journal data location to real home location. Our proposed method can prevent from degrading system performance while preserving file system consistency. Our approach is layered approach, which means that any existing file system can be setup upon our method with little modification to provide file system consistency. In the implementation, we consider Ext3 file system among many kinds of journaling file systems because it is the main root file system of Linux. The remainder of this paper is organized as follows. Section 2 describes motivation and background. Section 3 describes the design and implementation of our proposed technique, and Section 4 show experimental results. We conclude in Section 5.

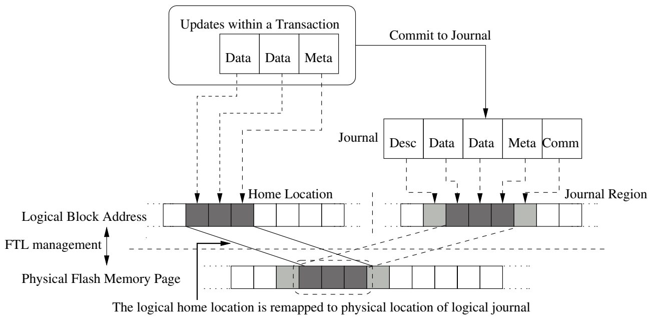
2 Background

Ext3 [13] is a modern root file system of Linux that aims to keep complex file system structures in a consistent state. All important file system updates are first written to a log region called a *journal*. After the write of journal are safely done, the normal write operations occur in its *home locations* of file system layout. The journal region is recycled after the normal updates is done. A central concept when considering a Ext3 file system is the transaction. A transaction is the update unit of Ext3 file system, which contains the sets of changes grouped together during an interval. A transaction exists in several phases over its lifetime, running, commit, checkpoint and recovery. The detailed operation of transaction is omitted due to the page limit. Please refer to other papers.

The Ext3 file system offers three journaling operation modes: *journaled mode*, *ordered mode*, and *writeback mode*. In *journaled mode*, all data and metadata are written to the journal. This provides high consistency semantics including user level data consistency. However, all are written twice to file system; first to the journal region, second to its home location. Thus, its write performance is extremely degraded. In *ordered mode*, only file's metadata are written to the journal, and file's data are written directly to its home location. To provide data consistency without double writes for file's data, *ordered mode* guarantees a strict ordering between two writes. The file's data are should be written to its home location before the corresponding metadata are written to the journal when a transaction commits. This can guarantee that file metadata never indicate a wrong data before it has been written. After the metadata logging to the journal is done, they are re-written to its home location. In *writeback mode*, as does *ordered mode*, only metadata are written to the journal, and file's data are



(a) Conventional FTL cannot eliminate double write


Updates within a Transaction
Data
Data
Meta
Commit to Journal
Journal
Desc
Data
Data
Meta
Comm
Home Location
Journal Region
Logical Block Address
FTL management
Physical Flash Memory Page
The logical home location is remapped to physical location of logical journal

(b) JFTL can eliminate double write by redirecting logical path of journal region to home location

Fig. 2. Double writes of Journaling

written directly to its home location. However, unlike *ordered mode*, there is no strict write ordering between these two. A metadata indexes can point to the wrong data block which wasn't written to storage. It cannot guarantee data consistency, but the journal can restore metadata.

The journaling operation of Ext3 can preserve file system consistency, at some level for each mode. Regardless of the consistency level of each mode, the key concept is that important information is first written to the journal, and then written to home location. During these journaling operations, double writes occur. If Ext3 is mounted in flash memory based storage system, it is set up upon FTL since FTL presents a block device interface to file system. The issue we should take notice is system performance. Since write operation occurs twice, write performance is degraded, since double writing occurs by a log manner in the flash memory. The Figure 2(a) represents the problem of double writing in FTL. However, if we make use of the FTL mapping management for journaling technique, we can eliminate double writing overhead. Figure 2(b) shows the *JFTL* that remaps the logical home location to the physical location of journal that was written before. It can eliminate the redundant writes of home location.

3 Design and Implementation

In this section, we describe our design and implementation of *JFTL*. The design goal is to prevent from degrading system performance, while providing file system consistency semantics. Our *JFTL* can be applied to all modes in Ext3 journaling, we explain based on Ext3 *journaled mode*.

3.1 JFTL: Journaling Remap-Based FTL

In the Ext3 *journaled mode*, all updates including file data and metadata are first written to *journal region*, then copied to *home location*. Transaction provides system consistency by writing all together or not included in a transaction. This atomicity is possible to manage duplication between journal region and home location, because original data remained in home location are not corrupted by operations during transaction running. The duplication is crucial feature in journaling. In view of this, if we look into FTL, FTL already uses duplication by the mapping management. Every written blocks from file system is written to another new flash pages by FTL not over-written. The problem is that FTL has no idea about journaling. Specifically, it does not know what data are associated with transactions, and the relationship between home location and journal region. If we can manage these information within FTL, we can eliminate double writing. In Ext3, these important information is stored in block called *descriptor*. For a transaction, *descriptor* block contains list of addresses of home location for each subsequent blocks to be written to journal region. The list represents the blocks to be written to home location after the transaction commit. During transaction commit phase, these are locked by journal transaction. The *pdflush* daemon of Linux may write journaled data to flash after commit ends. Thus, all updated data are duplicated.

In the *JFTL*, blocks in the list of *descriptor* is not written again to flash memory actually, but the *JFTL* remaps the physical location of journal region to home location. For this, we make a dedicated interfaces between journaling module and *JFTL* to pass these information. The lists recorded in *descriptor* are passed through the interface from Ext3 journaling module to *JFTL*, and *JFTL* uses these for updating the mapping table associated with blocks recorded in the list. The overall write process of journal including our remapping of duplicated blocks is as follows.

At first, when a new transaction is started, all updates are added and clustered to some buffers, and these are linked and locked. In the Figure 3, the updates, whose block numbers are 1 and 3, are clustered into running transaction.

Next, when the transaction expires, it enters commit phase where the clustered buffers, made in running phase, are inserted into list, log list. These data are written to journal region with two additional special blocks, *descriptor* and *commit*. In the Figure 4, Journal region represents the list to be written to journal region. In the journal, both *descriptor* and *commit* block have a magic header and a sequence number to identify associated transaction. The *descriptor* block records the home addresses of the list by checking their address of journal

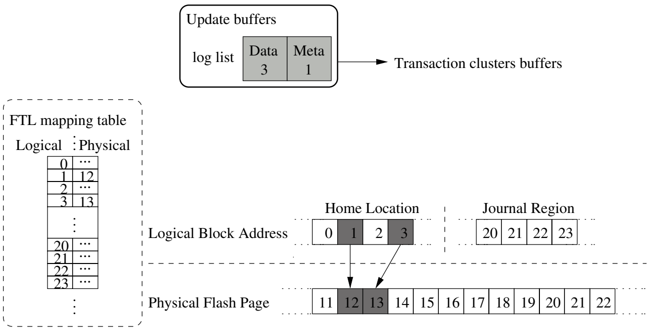


Fig. 3. In transaction running phase, buffers are clustered

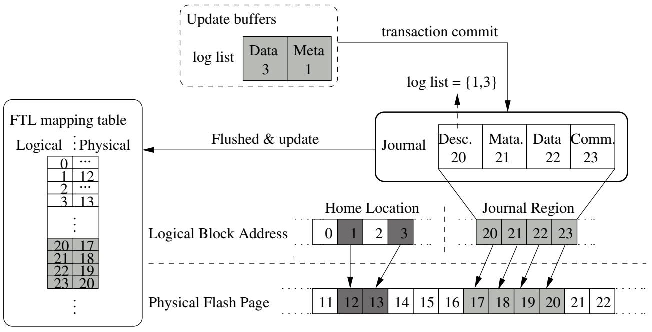


Fig. 4. Transaction commit phase makes journal and writes

region, according to the order of subsequent blocks that are written to journal region. For example, the block number of logging data within journal sequence is 1 and 3, so *descriptor* records the block number 1 and 3, which is associated to the journal logging sequence. When *descriptor* block is made with the list, blocks included in this transaction are flushed to block device layer. The submitted blocks are written to flash memory through the $JFTL$, and the $JFTL$ updates the mapping table. The region are logically journal region, but physically new free region. Up to now, journal data are flushed to storage.

After journal flushing is done, journaling module of Ext3 sends the list to $JFTL$ through the dedicated interfaces for remapping. $JFTL$ makes the remap list, *remap list*, from using information sent from journaling module. the *remap list* has not only home address but also journal address associated with it. For one element of the list is composed of home address and journal address to be updated, as shown in Figure 5. Specifically, the block number 1 of home location is related to the block number 21 of journal region, and the block number 3 of home location is related to the block number 22 of journal region. For each element, $JFTL$ finds out the physical address of the journal region, and updates mapping table of home location of the element with the physical address. Then,

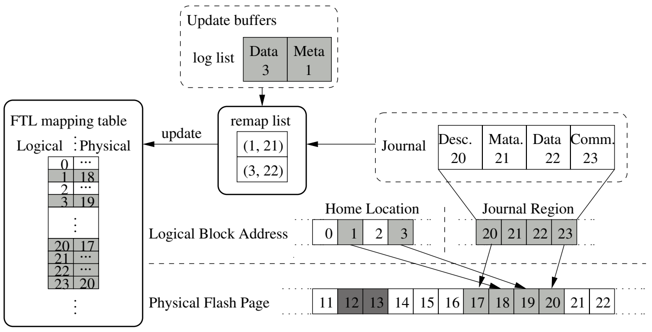


Fig. 5. JFTL remaps the home addresses of blocks to currently updated physical addresses

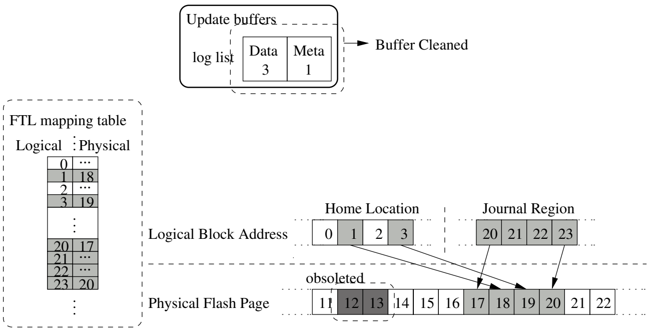


Fig. 6. Buffers are cleaned not to be flushed

JFTL clears the physical address of journal region. This clearing indicates that this region is obsoleted so it can be included in erase operation. In the Figure, in case of element (1,21), the physical address of journal region (21) is remapped to (18), so we link home location (1) to the physical address (18). Then we clear the physical address of journal region (21). When crash occurs before the remapping, this transaction journal data should be revoked and the file system status should be rolled back to the older version. This can be properly operated by recovering older version of mapping table. When crash occurs after the remapping, the new version is pointed by home location although this transaction is not properly completed. We don't worry about the consistency since the remapping can guarantee that all the journaling data in this transaction are written to storage.

When updating of mapping table associated to this transaction is done by remapping, the journaled data don't need to be written to flash memory any more, because all journal region are written out properly and the home location of that now points to the newly written data by remapping of these. The remaining job is to clear the buffers related to blocks of home location. This is possible by simply setting the BUFFER_CLEAN flag of $buffer\ head$ structure of the blocks. By doing this, these are not flushed to flash by the $pdflush$ daemon because it only flushed the dirty blocks by its mechanism. After all is done, the transaction is removed. Lastly, in $JFTL$, previous address of updated blocks now can be considered as obsoleted, so these are marked as dead in the table and can be included in erase operation.

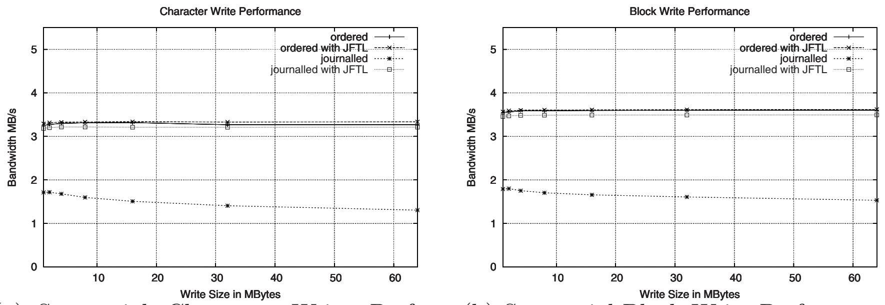
3.2 Recovery

When system crashes, Ext3 performs checking of its superblock and journal superblock, and scans the journal region to recover system corruption. When crash occurs during journal writes, we should provide rollback mechanism for improperly logged transaction. The improperly logged transaction represents that crash occurs during this outstanding transaction write. The recovery of Ext3 with $JFTL$ proceeds as follows. When Ext3 is mounted, it checks the mount flag in the superblock, and it finds that the system was crashed last. In this case, Ext3 checks journal superblock and identifies its consistency was corrupted by crash. After finding crashes, it identifies the committed transactions with their sequence number by scanning journal region. For the properly logged transactions, it reads *descriptor* block that is marked as same sequence number of superblock, checks the logged block addresses, and makes *remap list* for remapping these. The *remap list* is sent to $JFTL$, and $JFTL$ updates mapping table, as commit does during runtime. Then, Ext3 scans more to find logged transactions and repeatedly performs the updating processes as described above. During the checking of journal region, if Ext3 finds improperly committed transaction, it stops the recovery process and returns. Lastly, Ext3 resets the journal region to be cleared and used later for journaling.

Different from previous recovery process, in case of properly logged transaction, our recovery policy doesn't perform copies of logged data to permanent home location, only remaps their addresses to the physical position of that by updating $JFTL$ mapping table, just like that of run-time commit policy. Even though the amount of blocks to be recovered is small, our recovery mechanism is efficient because we only remap the committed transactions without copies to home locations.

4 Evaluation

The proposed Ext3 journaling mode with $JFTL$, will be compared to original Ext3 *ordered mode* and *journaled mode*. we exclude the *writeback mode* since it gives similar results of *ordered mode*. The experimental environment we use is NAND flash memory simulator based PC system. We use *nandsim* simulator [6] for the experiments. The nandsim can be configured with various NAND flash devices with associated physical characteristics. The physical characteristics of simulated the flash memory is as follows. Page size and block size of configured flash memory is 512Bytes and 16KBytes, respectively. One page read time and



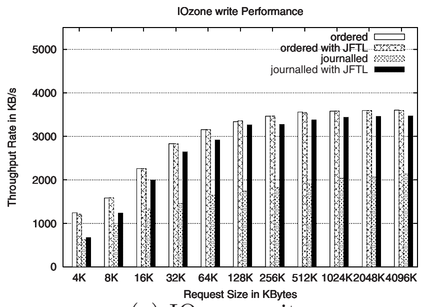
(a) Sequential Character Write Performance (b) Sequential Block Write Performance

Fig. 7. Bonnie Benchmark Performance. **The left graph plots character write performance, and the right graph plots block write performance as the amount of data written increases along the x-axis.**

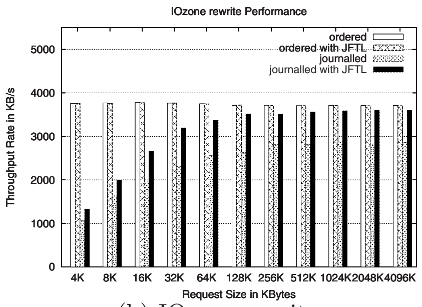
programming time is configured to 25us and 100us, respectively. Block erase time is set to 2ms. Those configured settings are accepted among many flash memories. The reason we chose 512Bytes page size is to match with the sector size of disk. The capacity of nandsim is set to 128MB, which is allocated in main memory. For the performance evaluation, we used two benchmark programs, Bonnie [15] and IOzone [16]. For the benchmark tests, we have tested as follows, Ext3 ordered, Ext3 ordered with $JFTL$, Ext3 journaled, and Ext3 journaled with $JFTL$. Among many file system operations, we are interested in write performance of file system since journaling gives overheads for write performance. During all the experiments, we guarantee all the data are written to flash memory, not be cached to main memory.

We begin our performance evaluation with Bonnie benchmark [15]. Bonnie performs a series of tests one file of known size. The series of tests are composed of character write/read, block write/read/rewrite, and random write. Among these, we plots character write and block write performance as file size increases. The Figure 7 shows their results. First of all, we can identify throughout the graphs that the bandwidth of each mode does not changed largely as the written file size increases. It is from distinct feature of flash memory. Flash memory is electrical device and does not affected by any geometry structure. Thus, flash memory is free from complex scheduling overhead like disk scheduling. Instead of, flash memory is affected by garbage collection issue.

The left graph of Figure 7 plots character write performance, and the right graph of Figure 7 represents block write performance of Bonnie. The character write operation indicates that file is written using the $putc()$ stdio. In block writes, file is written using $write()$ calls with the amount of block size, 4KB. They show similar results of write trends without average bandwidth rate values. In the graphs, among two ordered modes, the ordered with $JFTL$ slightly outperforms the ordered mode. It is the effect of remapping of metadata not real writing to home location. However, compared to large amount of data, the



(a) IOzone write



(b) IOzone rewrite

Fig. 8. IOzone Benchmark Performance. **The Left graph plots write performance, and the right graph plots rewrite performance as write request size per write operation increases along the x-axis.**

amount of metadata is negligible, so the performance gap is small. If we compare two journaled modes, journaled mode with $JFTL$ greatly outperforms the pure journaled mode, as we expect. In case of journaled mode, bandwidth is slowly down as file size increases, which is due to the number of descriptors in the journal transactions. As the written data is increased within a transaction interval, number of descriptors is also increased. If journaled with $JFTL$ is compared with ordered modes, bandwidth degradation of the journaled with $JFTL$ is not a serious problem. The journaled with $JFTL$ mode can give similar performance to ordered mode while preserving strict data consistency.

Next benchmark program we tested is IOzone benchmark [16]. It generates and measures a variety of file operations including write/rewrite, read/reread, aio read/write, and so on. Among these, we plot write performance verifying that data is actually written to flash memory. we have experimented write operation while varying the write request size per write operation from 4KBytes to 4096KBytes when file, whose size is 4096KBytes, is created and written. It means that 1024 write operation occurs for 4KBytes write size, whereas only one write operation occurs for 4KBytes write size, when 4096KBytes file is written. In the Figure 8, left graph plots write performance, and right graph plots rewrite performance as write size per write operation increases along the x-axis. The difference between write and rewrite is that write means new data is written and rewrite means the update of previous data. From the left graph, we identify that throughput of new data increases as the write size increases, which is due to the metadata update between each write intervals, such as free block allocation. For small write request size, performance is much degraded in both ordered mode and journaled mode. It results from the fact that free block allocation and metadata operation is performed for each write operation. As write request size increases, throughput is increased since metadata operation number is decreased. However, it is not required to allocate free blocks when the data is rewritten, which means metadata operation decreases. Thus, throughput of ordered mode is constant through the request sizes, as we see from the right graph. In all experiments, we can identify that the performance of ordered with $JFTL$ mode and journaled with $JFTL$ outperforms its comparison.

From the experiments, we note several results. First, journaling itself gives overhead to system while providing consistency. Second, when *ordered mode* is used, the performance of $JFTL$ is slightly better than original with the help of remapping. Third, the *journaled mode* downs system performance extremely, at least half of the throughput and latency, even if it provides strict data consistency. Whereas, the journaled with $JFTL$ does not harm largely, and gives little overhead to the system, while providing strict data consistency.

5 Conclusion

Flash Memory has become one of the major components of data storage since its capacity has been increased dramatically during last few years. Although it has many advantages for data storage, many operations are hampered by its physical characteristics. The major drawbacks of flash memory are that bits can only be cleared by erasing a large block of memory and the erasing count of each block has a limited number. Considering a journaling file system on the FTL, the duplication of data between the journal region and its home location is crucial for the file system consistency. However, the duplication degrades the file system performance.

In this paper, we present an efficient journaling interface between file system and flash memory. The proposed FTL, called $JFTL$, eliminates the redundant data copy by remapping the logical journal data location to real home location. Our proposed method can prevent from degrading system performance while preserving file system consistency. Our approach is layered approach, which means that any existing file system can be setup upon our method with little modification to provide file system consistency. In the implementation, we consider Ext3 file system among many kinds of journaling file systems because it is the main root file system of Linux.

References

1. Douglis, F., Caceres, R., Kaashoek, F., Li, K., Marsh, B., Tauber, J.A.: Storage alternatives for mobile computers. In: Proc. of the 1st Symposium on Operating Systems Design and Implementation(OSDI), pp. 25–37 (1994)
2. Best, S.: JFS Overview (2004), `http://www.ibm.com/developers-orks/library/l-jfs.html`
3. Sanmsung Electronics Co., NAND Flash Memory & SmartMedia Data Book (2002), `http://www.samsung.com/`
4. Ban, A.: Flash file system. U.S. Patent 5404485 (Arpil 4, 1995)
5. Intel Corporation: Understanding the flash translation layer(FTL) specification, `http://developer.intel.com/`
6. Memory Technology Device (MTD) subsystem for Linux, `http://www.linux-mtd.infradead.org`

7. Gal, E., Toledo, S.: Mapping Structures for Flash Memories: Techniques and Open Problems. In: Proceedings of the IEEE International Conference on Software-Science, Technology and Engineering (2005)
8. Woodhouse, D.: JFFS: The Journalling Flash File System. Ottawa Linux Symposium (2001)
9. Aleph One Ltd: Yaffs: A NAND-Flash File system, `http://www.aleph1.co.uk/yaffs/`
10. Lim, S.-H., Park, K.H.: An Efficient NAND Flash File System for Flash Memory Storage. IEEE Transactions on Computers 55(7), 906–912 (2006)
11. Kawaguchi, A., Nishioka, S., Motoda, H.: A Flash-Memory Based File System. Usenix Technical Conference (1995)
12. Rosenblum, M., Ousterhout, J.K.: The Design and Implementation of a Log-Structured File System. ACM Transactions on Computer Systems 10(1) (1992)
13. Ts'o, T., Tweedie, S.: Future Directions for the Ext2/3 File system. In: Proceedings of the USENIX Annual Technical Conference(FREENIX Track) (June 2002)
14. Timothy, E., et al.: Journal-guided Resynchronization for Software RAID. In: Proceedings of the 4th USENIX Conference on File And Storage Technologies(FAST) (December 2005)
15. Tim Bray: Bonnie Benchmark, `http://textuality.com/bonnie/`
16. William, D.: Norcott: Ioznoe File system Benchmark, `http://www.iozone.org/`