LTFTL: Lightweight Time-shift Flash Translation Layer for Flash Memory based Embedded Storage

Kyoungmoon Sun
School of Computer Sci. and Eng.
Dankook University
Yongin, Korea
msg2me@dankook.ac.kr

Seungjae Baek
School of Computer Sci. and Eng.
Dankook University
Yongin, Korea
ibanez1383@dankook.ac.kr

Jongmoo Choi
School of Computer Sci. and Eng.
Dankook University
Yongin, Korea
choijm@dankook.ac.kr

Donghee Lee
Dept. of Computer Science
University of Seoul
Seoul, Korea
dhl_express@uos.ac.kr

Sam H. Noh
School of Computer and Info. Eng.
Hongik University
Seoul, Korea
samhnoh@hongik.ac.kr

Sang Lyul Min
School of CSE
Seoul National University
Seoul, Korea
symin@archi.snu.ac.kr

ABSTRACT

Flash memory storage has been widely used in various embedded systems such as digital cameras, MP3 players, cellular phones, and DMB devices and now it applies to PCs as a form of SSDs. Characteristics of Flash memory necessitate a software layer called FTL (Flash Translation Layer) that directs modified data to new places in Flash memory and maintains a mapping between a logical sector number to a physical page. We notice that this out-of-place update scheme of the FTL allows a low-overhead time-shifting between multiple versions of storage state. From this observation, we propose LTFTL (Lightweight Time-shift FTL) that provides not only multiple versions of storage state but also an open-ended interface to traverse them. This open-ended interface can be used to support fault-resilience schemes, transactions of various granularities, and user-friendly roll-back services. Experimental results from a prototype implementation show that the proposed LTFTL can (1) provide a low-overhead time-shift capability at the user level by maintaining multiple storage states and (2) enhance the reliability/survivability of Flash memory by allowing to roll back to a previous consistent storage state at the storage system level.

Categories and Subject Descriptors

C.5.3 [**Microcomputers**]: Portable Devices; D.4.5 [**Operating System**]: Storage Management—Secondary storage

General Terms

Design, Performance, Reliability, Experimentation

Keywords

Flash memory, FTL (Flash Translation Layer), File system, Time-shift, Fault tolerance, Reliability

1. INTRODUCTION

Flash memory has advantages over conventional magnetic disks in terms of access time, power consumption, shock resistance, and weight. Hence, many embedded systems including cellular phones, digital cameras, MP3 players, and SSDs (solid state disks) utilize Flash memory as their storage medium. However, Flash memory has some limitations such as no in-place update and a limited life-time of its memory cells [1].

To overcome the limitations, most Flash memory software uses an out-of-place update scheme [1, 2, 3]. For updates of existing data, such a scheme allocates a new page, writes the new data to the allocated page, and invalidates the page that contains the (now obsolete) original data. The invalidated page is recycled after it is cleared.

Various schemes have been proposed for performing out-of-place update. In most of them, the focus was on minimizing the overheads associated with the out-of-place update [4 – 9]. In this paper, we contend that although we cannot avoid the inevitable costs of the out-of-place update, this same feature provides new opportunities to enhance the reliability of Flash memory-based storage device with only negligible additional overheads.

Based on the observation above, in this paper, we proposes a new FTL, called LTFTL (Lightweight Time-shift FTL) that supports a time-shift capability. It provides new interfaces to build up a consistent state of Flash memory at any time and to transit from one state to another. In this paper, the state of Flash memory is defined as a set of valid pages and transitions are expressed as a set of updating logs. By taking the merit of out-of-place update, we can implement the capability with only a marginal degradation of performance.

The time-shift capability can be exploited usefully in several fault-resilience mechanisms. For example, when a file system mount failure occurs we can try to mount with the most recent consistent state of Flash memory, which enhances the reliability/survivability of embedded storages. Also, it enables transactional operations with various granularities that support atomicity and recovery from crashes. Furthermore, it allows various version-control mechanisms, undelete facilities, and user-friendly roll-back services.

The proposed LTFTL has been implemented on an embedded system that has 64MB of NAND Flash memory running the Linux kernel 2.6.21. In the Linux kernel, Flash memory is managed by MTD (Memory Technology Device) layer [11] and this layer already has its own FTL, called NFTL [3]. We incorporated our LTFTL into this layer and compared the performance with the existing NFTL. Performance evaluation results show that LTFTL enhances the availability and consistency of Flash memory with minimal time and space overheads.

This paper is organized as follows. In Section 2, we describe characteristics of Flash memory and related works. Then, we introduce the time-shift capability and its model in Section 3. In Section 4, we discuss implementation details, and performance evaluation results are presented in Section 5. Finally, we provide a summary and directions for future works in Section 6.

2. FLASH MEMORY CHARACTERISTICS AND RELATED WORKS

In this section, we explain the structure and characteristics of Flash memory. Then, we describe why FTL is required and how it affects the performance of Flash memory.

2.1 Flash memory and FTL

Flash memory that is most widely used today is either NOR type or NAND type. One key difference between NOR and NAND Flash memory is the access granularity. NOR Flash memory supports byte-level random accesses, while NAND Flash memory supports only page-level accesses. Hence, in embedded systems, NOR Flash memory is typically used to store code, while NAND Flash memory is used as storage for the file system. From here we limit our focus to NAND Flash memory, though the ideas proposed in this paper, are also applicable to NOR Flash memory.

NAND Flash memory consists of same size blocks, each of which in turn consists of same size pages. The block size and the page size are typically 16 to 256 KB and 0.5 to 4 KB, respectively. Read/write operations are performed in page units, while the erase (i.e., clear) operation is performed in block units. One characteristics of Flash memory is the overwrite limitation – a block needs to be erased before new data can be written.

FTL is a software layer that supports out-of-place update to handle the overwrite limitation. Figure 1 shows two main functionalities of FTL: remapping and block cleaning. In the figure, we assume that each block consists of four pages and there is no written data in the Flash memory initially.

Figure 1(a) shows the contents of Flash memory after the initial write requests to the four logical sectors, numbered from 0 to 3 are serviced. These sectors are stored in pages, numbered from 0 to 3. Then, sectors 1 and 3 are requested to be updated, as shown in Figure 1(b). Due to the overwrite limitation, the update requests cannot be performed in-place. So, FTL allocates new pages (page 4 and 5 in the case of Figure 1(b)) and writes the updated data onto them. Then, FTL invalidates the original pages that previously contained the two sectors (pages 1 and 3) and modifies its mapping table to reflect these changes.

Block cleaning is a mechanism to reclaim the invalidated pages. It first chooses a block to be reclaimed and copies valid pages of the chosen block into another block that has been erased. Then, it erases the reclaimed block as shown in Figure 1(c). The mapping

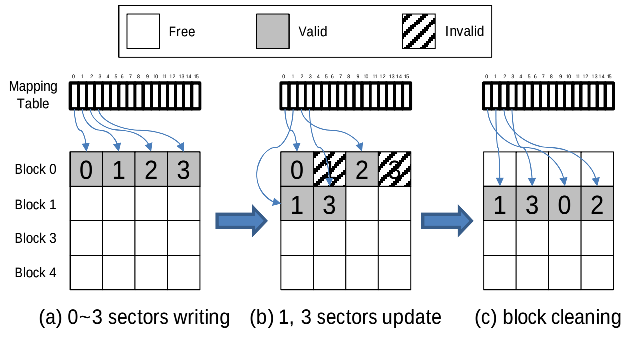

Figure 1. FTL: Remapping and Block Cleaning

and block cleaning mechanisms significantly affect the performance of Flash memory-based storage systems. Hence, much research on improving the effectiveness of these mechanisms has been performed in previous studies [1].

2.2 Related Works

According to the mapping unit, FTLs can be divided into three categories; page mapping, block mapping, and hybrid mapping. A page mapping FTL can map a logical sector into any physical page in Flash memory, allowing a more flexible management [2]. But, it carries a large space overhead for the mapping table. On the other hand, in a block mapping FTL, the mapping granularity is a block rather than a page [3]. It reduces the mapping table space overhead, compared with a page mapping FTL, but requires high overhead for block cleaning. A hybrid approach, called the *log block scheme* [4] and the *FAST scheme* [5] have been proposed to balance the space overheads and block cleaning overheads. Our proposed LTFTL is based on the page mapping mechanism.

For block cleaning, a page mapping FTL typically uses a garbage collection mechanism while a block mapping FTL uses a merge mechanism. The key issue of block cleaning is how to reduce the copy of the valid pages during reclamation. Kim et al. identified two merge mechanisms, a general merge and a switch merge, and showed that the switch merge can reclaim pages without any copy overheads [4]. Lee et al. proposed a new merge scheme that further reduces block cleaning overheads in the case of a repeated write pattern [9].

Like FTL, Flash memory file systems also contain remapping and block cleaning functionalities [6, 10, 13]. Kawaguchi et al. designed a page mapping flash memory file system and suggested a cost-benefit block cleaning mechanism [6]. Baek et al. proposed a new page allocation scheme that can reduce block cleaning overheads by improving the *uniformity* of a block [8]. Chiang el al. proposed a data clustering mechanism to allocate valid pages and invalid pages onto different blocks [7].

These studies mainly focus on the mapping and block cleaning mechanisms to reduce the performance degradation due to the out-of-place update characteristic. However, our study focuses on how to enhance the reliability by making use of the out-of-place update characteristic of Flash memory.

The reliability is a critical issue in Flash memory-based storage systems as well as in magnetic disks. Flash memory is widely used in embedded systems, especially in mobile consumer products that are prone to sudden power-failures resulting from abruptly detaching the battery. Also, Flash memory has inherent faults such as the bad block error, the bit flip error, and the wear limit [14]. Recently, MLC (Multi Level Cell) Flash memory where a single cell can represent multiple bits is employed widely to make high density embedded storage [15]. Since, in MLC Flash memory, the possibility of the bit flip error is higher and the number of erasures possible (i.e., wear-limit) on each block is more limited, the requirement of fault tolerance becomes indispensible.

FTLs and Flash memory file systems have their own mechanisms to handle bad blocks [2, 3, 10, 21]. A common approach is reserving some spare blocks and swapping the spare block with the bad block. Kim et al. showed that the bit flip errors occurring in Flash memory could make the whole file system inaccessible [16]. Chang proposed a wear-leveling mechanism that tries to reduce the wearing out of more worn out blocks by storing cold data in them, while storing hot data in less worn out blocks [17]. Agrawal et al. suggested a new wear-leveling algorithm controlled by two parameters, namely, *retirementAge* and *ageVariance,* and showed its effectiveness for extending SSD lifetime [25].

Gal et al. designed a transactional file system that supports atomic operations and recovery from crashes [18]. They support transactions in the file system layer with pruned versioned trees, whereas we support them in the FTL layer with time-shift capability. Lim et al. proposed a journal remapped-based FTL that can reduce the copy overheads by remapping the journal region to a home location in the ext3 file system [26]. Park et al. proposed an atomic write FTL that provides *AtomicWriteStart*() and *AtomicWriteCommit*() interfaces for robust flash file systems [27].

In the disk-based storage systems, the time-shift capability has been proposed in [19, 20]. Peterson et al. suggested a copy-on-write based file versioning and snapshot platform, called *ext3cow* [19]. Ta-Shma proposed a virtual machine based time travel scheme that enables reverts of the storage state to a point in the past by using a *live-migration based* checkpointing [20]. However, most disk-based storage systems use in-place update, so additional overheads are inevitable to provide such functionalities.

3. Time-shift capability in Flash memory

In this section, we first illustrate the concept of the time-shift capability. Then, we define the state of Flash memory and discuss a model for expressing the transitions between Flash memory states.

3.1 Basic idea

Figure 2 shows a write reference pattern and the contents of Flash memory. In the figure, we assume that write requests to sectors 0, 1, 2, 3, 0, 1, 4, 5, 3, 4, 5, 6 occur at time *t* $(0 \le t \le 11)$ where in this paper time is defined as a virtual time that is incremented on each sector write.

Figure 2(a) shows the contents of Flash memory at time $t=4$, when write requests to sectors from 0 to 3 are stored in pages 0 to 3. Also, Figure 2(b) and (c) show the contents of Flash memory at time $t=8$ and $t=12$, respectively.

At time $t=12$, we can define the state of Flash memory as follows.

$$S_{12} = \{P_0^4, P_1^5, P_2^2, P_3^8, P_4^9, P_5^{10}, P_6^{11}\}$$

– S_t : Flash memory state at time t.

– P_i^j : Data of sector number i is stored in page number j.

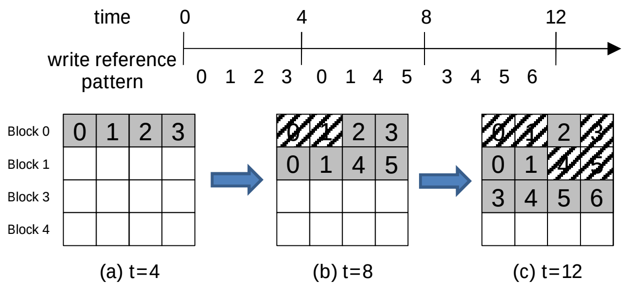

Figure 2. Write Reference Pattern and Flash Memory States

S_{12} represents the state of Flash memory at time $t=12$, where the sectors numbered from 0 to 6 are stored in pages numbered 4, 5, 2, 8, 9, 10, and 11, respectively.

Similarly, at time $t=8$, we can define the state of Flash memory as follows.

$$S_8 = \{P_0^4, P_1^5, P_2^2, P_3^3, P_4^6, P_5^7\}$$

It denotes that the state of Flash memory at time $t=8$ is the sectors numbered from 0 to 5 that are stored in pages numbered 4, 5, 2, 3, 6, and 7, respectively. Note that, by virtue of out-of-place update, even at time $t=12$, the pages for reconstructing S_8 are all still in Flash memory. In other words, at time $t=12$, we can revert from S_{12} to S_8, which is the basic idea of our proposed time-shift capability in Flash memory.

The time-shift from S_{12} to S_8 is expressed as follows.

$$S_8 = (S_{12} \cup Domain(U_{8\sim12})) - Range(U_{8\sim12})$$

– $U_{8\sim12}$: Set of mapping information update during the period $8\sim12$.

– $Domain(R)$: Set of first coordinate values of R.

– $Range(R)$: Set of second coordinate values of R.

In this example of Figure 2, *U, Domain(U),* and *Range(U)* are expressed as follows.

$$U_{8\sim12} = \{(P_3^3, P_3^8), (P_4^6, P_4^9), (P_5^7, P_5^{10}), (\phi, P_6^{11})\}$$

$$Domain\ (U_{8\sim12}) = \{P_3^3, P_4^6, P_5^7\}$$

$$Range\ (U_{8\sim12}) = \{P_3^8, P_4^9, P_5^{10}, P_6^{11}\}$$

The above equations imply that by maintaining the update information, we can revert from the current state into any previous states (of course, we need to control block cleaning mechanisms as explained in Section 3.2).

3.2 Model

Formally, the time-shift from S_t to $S_{t'}$ can be expressed as follows.

$$S_{t'} = (S_t \cup Domain(U_{t'\sim t})) - Range(U_{t'\sim t}) \quad \text{--- (1)}$$

In Equation (1), $S_{t'}$ and S_t are the states of Flash memory at time t' and t, respectively. Each Flash memory state is defined as a set of valid pages at the time. The term $U_{t'\sim t}$ represents the set of modified mappings between t' and t. Then, $Domain(U_{t'\sim t})$ is defined as the set of the original mappings, while $Range(U_{t'\sim t})$ is defined as the set of the new mappings. The null element (ϕ) in

Domain($U_{t'\sim t}$) denotes that a mapping for a logical sector that has not been mapped to any page is newly established.

Now, the question is how to design a block cleaning mechanism for the time-shift capability. Conventionally, FTL reclaims all invalid pages during block cleaning. However, in LTFTL, we need to take into account the pages that should be reserved for the time-shift capability. Figure 3 shows the state of each page in conventional FTL and our proposed LTFTL.

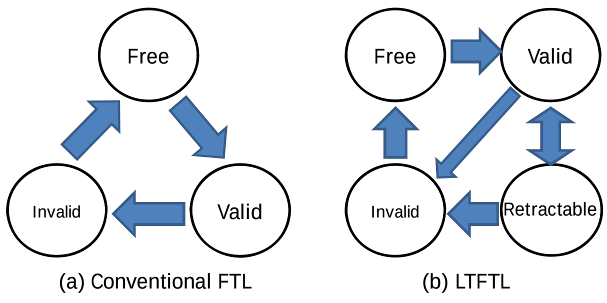

Figure 3. Page Status in Conventional FTL and LTFTL

In conventional FTL, each page is in one of three possible states; free, valid, and invalid. If a page holds legitimate data, it is a valid page. When a valid page is invalidated by being deleted or by being updated, then the page become an invalid page and is treated as a candidate for block cleaning. Finally, by performing block cleaning, the pages that reside on blocks that have been erased become free and eligible to be written to.

In LTFTL, some invalid pages should not be subject to block cleaning so as to provide the time-shift capability. We define these pages as *retractable* pages. Let us assume that the current time is *t* and we might revert to the previous state of Flash memory at time *t'*. Then, the retractable pages can be expressed as follows.

$$R_t = \{S_{t'} \cap Domain(U_{t'\sim t})\} \quad \text{----} \ (2)$$

Equation (2) implies that, among the invalid pages, the pages that belong to the state of Flash memory specified as revertible become retractable. If there are several revertible states, the union of each R_t become the overall retractable pages. In LTFTL, the block cleaning mechanism can reclaim invalid pages that are not retractable. From this discussion, we find that there are trade-offs between the number of revertible states the user can specify and the space overhead for keeping retractable pages.

4. Implementation Details

In this section, we explain how we implemented the proposed LTFTL in the Linux kernel.

Figure 4 shows the software architecture of the Linux kernel for Flash memory. It consists of three layers; VFS (Virtual File System) layer, specific file system layer (such as Ext3, FAT, JFFS2, and YAFFS), and MTD (Memory Technology Device) layer. The MTD layer is further divided into three layers; MTD user module layer, MTD glue logic layer, and MTD chip driver layer. The MTD user module layer includes some open source FTLs such as NFTL and FTL. NFTL is a block mapping FTL for

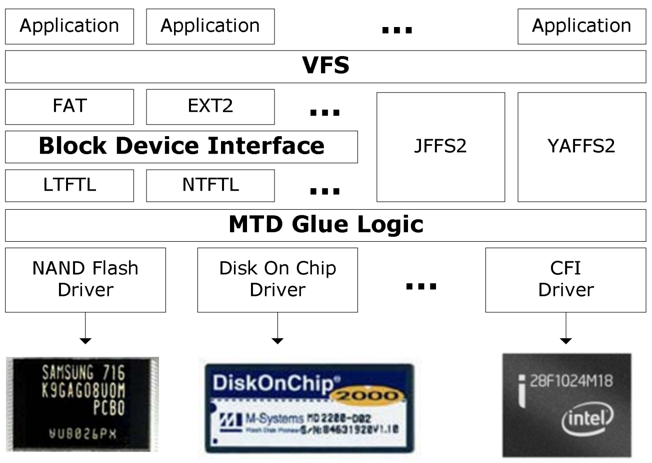

Figure 4. Flash Memory Software Architecture in Linux

NAND Flash memory that uses a block chain mechanism to support out-of-place update [3]. FTL is a page mapping FTL that only supports NOR Flash memory [2].

We first tried to modify NFTL to implement our time-shift capability. However, it is very difficult to support the lightweight time-shift capability in the NFTL framework, mainly due to its block mapping nature. Therefore, we decided to implement our own LTFTL that uses a page mapping mechanism. Our LTFTL resides in the MTD user module layer, just like NFTL. Any file system that can run on top of hard disks such as the FAT and Ext3 file systems can run on top of either NFTL or LTFTL (note that Flash memory file systems such as YAFFS and JFFS2 that run directly on Flash memory without FTL support do not belong to this group).

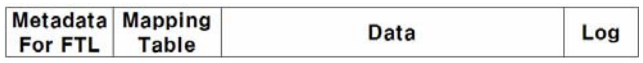

Figure 5. Logical Layout of LTFTL

Figure 5 shows the logical layout of LTFTL. It partitions Flash memory into four sections; metadata, mapping, data, and log sections. The metadata section contains control information of LTFTL such as the location of each section, the current Flash memory state id, and the oldest revertible Flash memory state id. The mapping section contains the mapping table that translates logical sector numbers into physical page numbers. The data section contains data for logical sectors and is managed by page-mapping and garbage-collected. Finally, the log section contains the intent log that keeps information for each write request including the original page, the new page, and the requested sector number.

The mapping table uses an inverted mapping structure, where each entry in the table indexed by a page number contains the corresponding sector number. During the LTFTL initialization, the forward mapping structure that maps a sector number into the corresponding page number is created on SDRAM and used for actual address translations. The reason for using the inverted structure for the in-Flash mapping table is that we want to use the table not only for translating between sector numbers and page numbers but also for identifying the state of each page. For example, an entry of the table holding a valid sector number implies that its state is valid, while an entry with 0xffff ffff represents its state as free. The overall size of the mapping table is 4B × sizeof(Flash memory) / sizeof(page).

The data structure for each entry in the log section is defined as <sector number, old page number, new page number, global_time>. The global_time is incremented at each write request and used as the sequence number to manage a system wide virtual time and to differentiate between old and new pages. We define each field in the data structure as integer type and the size of the log section is changed dynamically depending on the number of log entries that in turn depends on the number of revertible states.

In LTFTL, the log section plays two roles. The first is maintaining a transition of $U_{t'\sim t}$ between two states $S_{t'}$ and S_t defined in Equation (1). The global_time is used for *t'* and *t* in this case. The second role is to allow a delayed update of the in-Flash mapping table without loss of data even under power failures. Each log entry is a record of intent for updates to be performed on Flash memory. Hence, if we keep all log entries, we can generate the up-to-date mapping table from any old mapping table.

As a whole, the LTFTL handles write requests as follows. It first allocates a new page from the data section and writes the requested data on the page. Then, it modifies the forward mapping structure on SDRAM to reflect this change. Finally, it adds a log entry into the log section. In the implementation, we generate the up-to-date mapping table and store it in the mapping section when LTFTL is initialized or when the size of the log section become larger than a maximum threshold (we set the threshold to 16B × sizeof(Flash memory) / sizeof(page) per each Flash memory state, in the current implementation).

Note that, in NAND Flash memory, writing is performed in page units. So, we use one page size of SDRAM as a write buffer for collecting log entries and store them on the log section when the write buffer is full or when the system is normally shutdown. Now the question is how to protect the contents of the write buffer from a sudden power failure? We piggyback <sector number, global_time> onto each write request and store them in the OOB(Out Of Band) area, also called the spare area, of the page allocated for the request. During the initialization of LTFTL, we can detect the abnormal shutdown of the system from a flag recorded in the metadata section. Then, we scan the OOB of the whole pages and reconstruct log entries for the write buffer by identifying the OOB whose global_time is larger than the latest one stored in the log section.

For the time-shift capability, LTFTL provides three interfaces; LTFTL_freeze(), LTFTL_unfreeze(), and LTFTL_revert(). The LTFTL_freeze() interface is used to build a new state of Flash memory at the current time. It returns a state_id, which is used as an argument for subsequent LTFTL_unfreeze() and LTFTL_revert() requests that are explained next. The LTFTL_unfreeze() interface is used to remove a Flash memory state. Finally, LTFTL_revert() is used to change from the current state to the state specified as an argument.

A more detailed description of the processing of LTFTL_freeze() is as follows: (1) allocate space from the metadata section for managing the information of the new state; (2) set the 'global_time of the last log entry' as the epoch time of the state; (3) set every subsequently invalidated pages as retractable pages related to the state. The LTFTL_unfreeze() interface is implemented as follows: (1) set the associated retractable pages of the state as invalid pages; (2) remove the log entries for the state if they are not needed any more for other states and for generating the up-to-date mapping table; (3) free the allocated space for managing the state information. Finally, the LTFTL_revert() interface is implemented as follows: (1) generate the up-to-date mapping table; (2) roll back to the specified state by undoing the operation of the log entries backward in time until it reaches the epoch time of the state.

The garbage collection mechanism used by LTFTL to reclaim the invalid pages is similar to those for other page-mapping FTLs except for the handling of pages with retractable state - when there are not enough free pages, the mechanism first selects blocks that have higher ratios of invalid pages to the total pages in the block and performs block cleaning. Also, we employ a simple wear-leveling mechanism that periodically switches blocks among the sections. Designing more efficient algorithms for garbage collection and wear-leveling for LTFTL are left as future work.

5. Performance Evaluation

In this section, we first present the time and space overheads of the proposed LTFTL. Then, we present evaluation results to assess the impact of LTFTL on the reliability of Flash memory.

5.1 Overhead Analysis

We have implemented LTFTL on an embedded system. The hardware components of the system include a 400MHz XScale PXA CPU, 64MB SDRAM, 64MB NAND Flash memory (page size: 512B and block size: 16KB), and embedded controllers such as those for LCD, UART and JTAG [22]. The system is running the Linux kernel 2.6.21 and our LTFTL is implemented in the MTD user module layer. Then, we run the FAT file system on LTFTL. For comparison purposes, we also run the FAT file system on the existing NFTL.

```
localhost ~ # mount /dev/ltftla1 mnt
localhost ~ # echo "It's an original string" > mnt/file
localhost ~ # ltftl_freeze
LTFTL freeze: 10
localhost ~ # echo "It's a modified string" > mnt/file
localhost ~ # cat mnt/file
It's a modified string
...
localhost ~ # ./ltftl_revert 10
localhost ~ # cat mnt/file
It's an original string
localhost ~ #
```

Figure 6. Example of Multiple States

Figure 6 shows an example of time-shifting supported by LTFTL. We first create a file whose content is "It's an original string", and build a state using the *ltftl_freeze* command. The command invokes the LTFTL_freeze() interface internally to perform the requested function. Then, we change the content of the file as “It’s a modified string”. Later, we want to restore the original contents of the file and issue the *ltftl_revert* command that calls the LTFTL_revert() interface. After the revert operation, we find that the image of the file system is actually time-shifted to the state we specify.

To evaluate the overhead of LTFTL, we ran the Postmark benchmark [23] and measured the execution time of the benchmark as shown in Figure 7 while invoking different numbers of the LTFTL_freeze() function. The benchmark creates a large number of randomly sized files. It then executes read, write, delete, and append transactions on these files. In this experiment, we created 1000 files ranging in size from 512byte to 9.77KB and performed 38000 transactions that made roughly 32MB of Flash memory as valid pages. In the figure, NFTL represents the default FTL of the MTD in the Linux kernel. LTFTL(none) represents the case of LTFTL that does not invoke the LTFTL_freeze() interface during the benchmark execution, while LTFTL(*n*) represents the case that the interface is invoked *n* times.

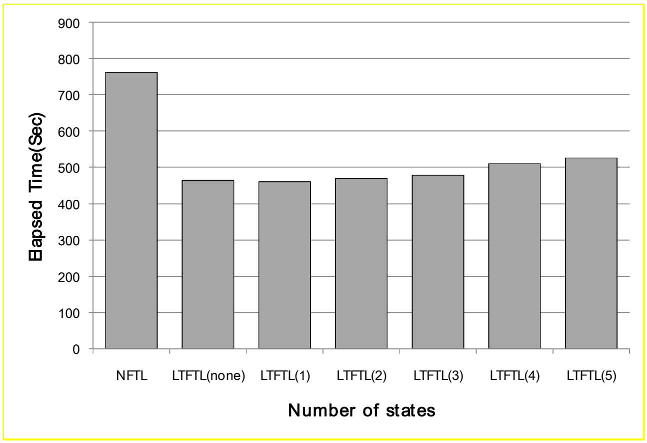

Figure 7. Execution time for Postmark

From Figure 7, we find that LTFTL performs comparably with the existing NFTL. Although a direct comparison between NFTL and LTFTL is unfair since the former employs a block level mapping while the latter a page level mapping, the results show that our implementation of LTFTL does not incur excessive overheads. A more detailed experiment shows that the block chain scheme used by NFTL incurs frequent block cleaning for repeated small writes, which results in an inferior performance as compared against LTFTL.

As the number *n* in LTFTL(*n*) increases, the overhead of LTFTL also increases. For example, the execution time of LTFTL(1) is 460 seconds while that of LTFTL(5) is 526 seconds. This additional overhead includes the housekeeping costs for log entries and the state information management. Also, it includes the increased block cleaning costs due to the keeping of retractable pages. As a result, as *n* increases, the block cleaning is performed more frequently since the number of free pages reclaimed in each block cleaning is reduced. Such trends are evident in Figure 8, which shows the usage patterns of Flash memory for each FTL.

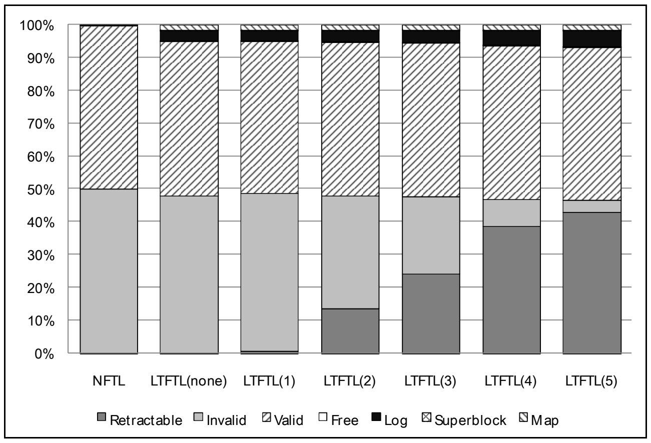

Figure 8. Flash Memory Usage Patterns

Figure 8 shows that in NFTL 49.72% of Flash memory is occupied by valid pages while 50.17% is occupied by invalid pages. The remaining 0.12% is used by metadata, map, and free pages. In LTFTL(1), 46.51%, 47.91%, 2.98%, 1.59%, and 0.76% of Flash memory are used by valid, invalid, log, map, and retractable pages, respectively. In contrast, in LTFTL(5), 46.87%, 3.53%, 4.77%, 1.59%, and 43.01% of Flash memory are used by valid, invalid, log, map, and retractable pages, respectively. In LTFTL(5), block cleaning can reclaim only a limited number of invalid pages, which makes LTFTL(5) trigger block cleaning more frequently. We expect that, in most cases, LTFTL can support various fault-resilience mechanisms with only 1 or 2 revertible states of Flash memory. In such cases, since there are still plenty of invalid pages to reclaim, the execution time difference is not much (460, 469, and 487 seconds for LTFTL(0), LTFTL(1), and LTFTL(2), respectively).

5.2 Reliability Analysis

Table 1 investigates the impact of LTFTL on the reliability of Flash memory. To measure the reliability, we simulated sudden power failures, one of the most common type of faults in battery-operated mobile embedded systems. More specifically, we simulated a sudden power failure for each of 5060 sector writes generated by a Postmark benchmark consisting of 100 transactions to 100 files. The same experiments were performed for both NFTL and LTFTL and the reliability was measured in two different ways: (1) mount success ratio and (2) fsck (file system consistency checker) pass ratio. In the case of LTFTL, we performed LTFTL_freeze() when (1) the total amount of storage write since the last LTFTL_freeze() exceeds 25 KB and (2) there are no file system activities. Also, immediately after the LTFTL_freeze(), we performed LTFTL_unfreeze() for the Flash memory state created by the LTFTL_freeze() before the current one. Thus, at all times there was only one revertible state that was guaranteed to be consistent since it was made when there were no file system activities and the LTFTL always returned to this state at the subsequent reboot time.

Table 1. Mount Success Ratio and fsck Pass Ratio on a Sudden Power Failure

	Mount Success Ratio	fsck Pass Ratio
LTFTL	100% (5060/5060)	100% (5060/5060)
NFTL	99.72% (5046/5060)	56.75% (2872/5060)

Table 1 gives the two reliability measures for LTFTL and NFTL. As expected, the LTFTL gives a 100% mount success ratio. Even for NFTL, the mount success ratio is very high (99.72%). However, in the case of NFTL, even when the mount is successful, the file system still contains inconsistencies as we can see from its low fsck pass ratio (56.75%). But for LTFTL, the fsck pass ratio is also 100% since it always rolls back to a consistent Flash memory state on a power-failure recovery.

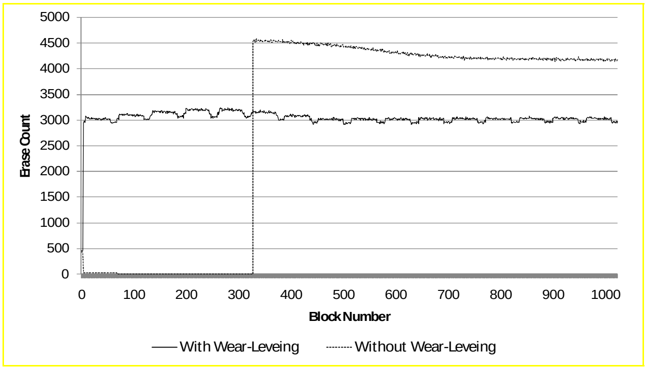

Figure 9. Wear-leveling Effects

Figure 9 shows the effects of the wear-leveling mechanism used by LTFTL. Recall that LTFTL partitions Flash memory into four sections as shown in Figure 5. This makes some blocks to be more worn out than others. To prevent this problem, after 50 erase operations, we select one block from each section (except for the metadata section) and perform a swap between them. Figure 9 shows that in LTFTL without such proactive wear-leveling, the erase count of the blocks from 0 to 350 is significantly lower than those over 350. We find out that the less worn out blocks are those for the mapping table and retractable pages. In LTFTL with wear-leveling, we can that blocks are evenly worn out since the block switching allows for blocks once used for the mapping table and retractable pages to be used for data blocks as well at other times.

6. CONCLUSION

Two contributions are made in this paper. First, we identify that the out-of-place update scheme of Flash memory can be exploited usefully to provide lightweight time-shift capability. Second, through an implementation study, we demonstrate that the lightweight time-shift capability can enhance the reliability of Flash memory by providing consistent Flash memory states to roll back.

We are considering two research directions for future work. One direction is to implement the time-shift capability in Flash memory file systems such as YAFFS [10]. A file system interacts with user applications directly, which provides more opportunities for the time-shift. Another direction is applying our model to LFS (Log-structured File System) [24]. Though LFS is a disk-based file system, it uses the out-of-place update scheme. We need to investigate how to manage intent logs in large scale storage and how to reduce the state transition overheads.

7. ACKNOWLEDGMENT

This work was partly supported by the IT R&D program of MKE/IITA [2006-S-040-03, Development of Flash Memory-based Embedded Multimedia Software] and by the Korea Science & Engineering Foundation (KOSEF) through the NRL Program (No. R0A-2007-000-20071-0). Also, this work was supported in part by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (No. R01-2008-000-12028-0).

8. REFERENCES

[1] E. Gal and S. Toledo, “Algorithms and Data Structures for Flash Memories”, ACM Computing Surveys, vol. 37, no. 2, pp. 138-163, 2005.

[2] Intel Corporation, “Understanding the Flash Translation Layer (FTL) Specification”, 1998.

[3] M-Systems, “Flash-Memory translation layer for NAND flash (NFTL)”, 1998.

[4] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient Flash translation layer for CompactFlash systems", IEEE Transactions on Consumer Electronics, vol. 48, no. 2, pp. 366-375, 2002.

[5] S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. Park, and H. J. Song, "A Log Buffer based Flash Translation Layer using Fully Associative Sector Translation", ACM Transactions on Embedded Computing Systems, vol. 6, issue 3, 2007.

[6] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-memory based file system", Proceedings of the 1995 USENIX Annual Technical Conference, pp. 155-164, 1995.

[7] M-L. Chiang, P. C. H. Lee, and R-C. Chang, "Using data clustering to improve cleaning performance for Flash memory", Software: Practice and Experience, vol. 29, no. 3, pp. 267-290, 1999.

[8] S. Baek, S. Ahn, J. Choi, D. Lee, and S. H. Noh, “Uniformity Improving Page Allocation for Flash Memory File system”, ACM International Conference on Embedded Software (EMSOFT), pp. 154-163, 2007.

[9] J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, and S. H. Noh, “Block Recycling Schemes and Their Cost-Based Optimization in NAND Flash Memory Based Storage system”, ACM International Conference on Embedded Software (EMSOFT), pp. 174-182, 2007.

[10] Aleph One, "YAFFS: Yet another Flash file system", www.aleph1.co.uk/yaffs/.

[11] MTD subsystem for Linux, http://www.linux-mtd.infradead.org/archive/index.html.

[12] Linux VFAT File System, http://bmrc.berkeley.edu/people/chaffee/vfat.html.

[13] S. Lim and K. Park, “An Efficient NAND Flash File System for Flash Memory Storage”, IEEE Transactions on Computers, Vol. 55, No. 7, July, 2006.

[14] Samsung Electronics, Co., “NAND Flash Memory and SmartMedia Data Book”, 2004.

[15] Samsung Electronics, Co., “1G x 8Bit / 2G x 8Bit NAND Flash memory (K9L8G08U0M) Data Sheets”, 2005

[16] S. Kim, J. Choi, D. Lee, S. H. Noh, and S. L. Min "Virtual Framework for Testing the System Software on Embedded System", ACM Symposiums on Applied Computing, pp. 1192-1196, 2007.

[17] Li-Pin Chang, “On Efficient Wear Leveling for Large-Scale Flash-Memory Storage Systems”, ACM Symposium on Applied Computing, pp.1126~1130, 2007.

[18] E. Gal and S. Toledo, "A transactions Flash file system for microcontrollers", Proceedings of the 2005 USENIX Annual Technical Conference, pp. 89-104, 2005.

[19] Z. Peterson and R. Burns, "Ext3cow: A Time-shifting File System for Regulatory Compliance", ACM Transactions on Storage, Volume 1 , Issue 2, pp. 190~212, 2005.

[20] P. Ta-Shma, G. Laden, M. Ben-Yehuda, M. Factor, “Virtual Machine Time Travel Using Continuous Data Protection and Checkpointing”, ACM Operating Systems Review, Vol. 42, Issue 1, Jan. 2008.

[21] D. Woodhouse, "JFFS: The journaling Flash file system", Ottawa Linux Symposium, 2001, http://source.redhat.com/jffs2/jffs2.pdf.

[22] EZ-X5, www.falinux.com/zproducts.

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

[24] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system", ACM Transactions on Computer Systems, vol. 10, no. 1, pp. 26-52, 1992.

[25] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse and R. Panigrahy, “Design Tradeoffs for SSD Performance”, Proceedings of the 2008 USENIX Annual Technical Conference, pp. 57-70, 2008.

[26] S. Lim, H. Choi and K. Park, “Journal Remap-Based FTL for Journaling File System with Flash Memory”, The 3rd International Conference on High Performance Computing and Communications, pp. 192-203, 2007.

[27] S. Park, J. Yu and S. Ohm, “Atomic Write FTL for Robust Flash File system”, Proceeding of the 9th International Symposium on Consumer Electronics, pp. 155-160, 2005.