A File-System-Aware FTL Design for Flash-Memory Storage Systems

Po-Liang Wu
Department of Computer Science and Information Engineering
National Taiwan University,
Taipei 106, Taiwan, R.O.C.
Email: b91029@csie.ntu.edu.tw

Yuan-Hao Chang
Department of Computer Science and Information Engineering
National Taiwan University,
Taipei 106, Taiwan, R.O.C.
Email: d93944006@csie.ntu.edu.tw

Tei-Wei Kuo
Department of Computer Science and Information Engineering
National Taiwan University,
Taipei 106, Taiwan, R.O.C.
Email: ktw@csie.ntu.edu.tw

***Abstract*—As flash memory became popular over various platforms, there is a strong demand on the performance degradation problem, due to the special characteristics of flash memory. This research proposes the design of a file-system-aware flash translation layer, in which a filter mechanism is designed to separate the access requests of file-system metadata and file contents for better performance. A recovery scheme is then proposed to maintain the integrity of a file system. The proposed flash translation layer is implemented as a Linux device driver and evaluated with respect to ext2 and ext3 file systems. The experimental results show significant performance improvement over ext2 and ext3 file systems with limited system overheads.**[1]

I. INTRODUCTION

The applications of flash memory technology have grown much beyond its original design goal. The success of flash memory technology not only comes with a significant market growth rate but also introduces new challenges. One popular example is a solid state disk (SSD)[2]. Although Single-Level-Cell (SLC) flash memory is more reliable, its cost weakens itself from the strong competition from Multi-Level-Cell (MLC) flash memory in storage-system designs[3]. As the capacity of storage system is doubled roughly every 18 months, we face challenging performance and reliability problems of MLC flash memory. For example, the time to program one page of $\text{MLC}_{\times 2}$ flash memory is $800\mu s$, and each block of $\text{MLC}_{\times 2}$ flash memory can only endure 10,000 erase cycles, compared to $200\mu s$ page-program time and 100,000 erase cycles of SLC flash memory [1], [2]. These problems are further exaggerated by the behavior of file systems (that are usually over storage systems), mainly due to frequent writes of small sizes to the flash memory [3]. This observation motivates this research on the exploring of file-system behaviors and the design of a file-system-aware flash-memory management scheme.

A NAND flash memory chip is organized in terms of blocks, and each block is further divided into a fixed number of pages. A block is the basic unit for erase operations, while reads and writes are processed in the unit of one page. The typical block size and the page size of a SLC(/MLC) NAND flash memory are 128KB(/256KB) and 2KB, respectively. A page can not be overwritten unless it is erased. Therefore, out-place-update is usually adopted in flash-memory management, where out-place update is to write to-be-updated data to another free page of flash memory. In the past decades, there were many excellent work and implementations in meeting the performance and reliability requirements of flash-memory storage systems, *e.g.*, [4], [5], [6], [7]. Some exploited different system architectures and layer designs, *e.g.*, [5], [8], [9], [10], and some explored the design issues of the native file systems over flash-memory storage devices, *e.g.*, [4], [8], [11]. Researchers also exploited large-scaled and energy-aware storage systems, *e.g.*, [12], [13]. In particular, Wu *et al.* [9] proposed a hybrid flash-translation-layer design to switch the mapping information between a fine-grained and a coarse-grained address translation mechanisms dynamically and adaptively. Hsieh *et al.* [14] proposed a caching strategy and data structure with the considerations of the characteristics of NAND flash memory and energy consumption of the system on processing read and write requests. Chang *et al.* [15] proposed a hybrid architecture to combine SLC and MLC NAND flash memory on Solid State Disks so as to utilize the advantages of SLC and MLC flash memory. A *Hot-Data Filter* is designed to identify frequently accessed data from non-frequently accessed data according to the size of accessed data in each request. Moreover, Samsung presented a OneNAND structure, which uses NAND flash memory and a SRAM buffer to replace NOR flash memory, *e.g.*, [16], [17], [18]. NAND flash memory is also adopted to facilitate the booting process or to save the energy consumption of hard disks [19], [20], [14].

However, due to the special write constraints of MLC flash memory, partial-programming to a page is impossible, and random writes to pages of a block are prohibited. These write constraints makes many existing flash-translation-layer (FTL) designs infeasible or lack of efficiency. Different from the previous work, we are interested in the performance and reliability improvements for different file systems with the consideration of flash-memory characteristics. In this paper, we propose the design of a file-system-aware flash translation layer with the considerations of file system behavior and reliability issues. An *efficient filter mechanism* is designed to separate the access requests of file metadata and contents for better performance based on the knowledge of the layout of file systems. A recovery scheme is also proposed to maintain

[1]This work is partially supported by the Genesys Logic, the National Science Council of Taiwan, and Excellent Research Projects of National Taiwan University, under Grant NSC-95-2221-E-002-094-MY3 and 97R0062-05

[2]There are two types of flash memory: NOR and NAND. In this paper, we consider NAND flash memory because it is the most widely adopted flash memory in storage systems.

[3]Each cell of MLC$\times n$ flash memory can store n-bit information, while each cell of SLC can store only one-bit information

978-3-9810801-5-5/DATE09 © 2009 EDAA

the integrity of a file system. The proposed flash translation layer is implemented as a Linux device driver on a *DaVinci* evaluation board [21] and evaluated with respect to ext2 and ext3 file systems.

The experimental results show that 20% performance improvement can be obtained for ext2/ext3 file systems. Moreover, the extra RAM consumption of the proposed FTL design over ext2/ext3 is under 2.4% of existing designs.

The rest of this paper is organized as follows: Section II presents the system architecture and the motivation of this work. Section III presents file-system-behavior analysis and proposes an efficient file-system-aware FTL design. Section IV summarizes the experimental results of the proposed FTL design over popular Linux file systems. Section V is the conclusion.

II. SYSTEM ARCHITECTURE AND RESEARCH MOTIVATION

A. System Architecture

Consider a typical system architecture of flash-memory-based storage systems, as shown in Figure 1, where a flash storage device is usually connected to a host system through a standard interface. In order to control flash media of a flash storage device, a Memory Technology Device (MTD) driver is needed to support primitive flash-memory operations, such as reads, writes, and erases. A Flash Translation Layer (FTL) driver is used to provide advanced functionality in address translation, garbage collection, and wear-leveling. Here address translation is to translate any given logical block addresses (LBAs) to physical block addresses (PBAs), where each LBA represents a storage unit of 512B. Note that data updates must be written to free pages of flash memory such that there is a need in address translation, and pages of old data versions become invalid after each update. Garbage collection is to reclaim space that store invalid data, whenever there are no sufficient free pages. In other words, garbage collection selects victim blocks that contain invalid pages, copies valid pages from the victim blocks to free pages (referred to as *live-page copyings*), and then erases the victim blocks. Wear-leveling is to distribute block erases as evenly as possible over flash-memory chips. An FTL driver usually emulates the underlying flash memory as a block device so that file systems can access the flash storage device transparently.

One increasing critical problem for users is that existing file systems, such as ext2 and ext3, are designed with little awareness of flash-memory characteristics.

It is mainly because they are designed when hard drives (that rely on in-place updates) are the most popular secondary storage devices. Such an observation underlines our research motivation in the investigation of the design issues of FTL drivers in reducing the potential overheads imposed by file systems on flash memory.

There are three major implementation approaches of flash-memory management schemes: *FTL*, *BL*, and *NFTL*: FTL adopts a page-level address translation mechanism for fine-grained address translation [22], [23]. It relies on an address translation table to map any given LBA to its PBA, where LBAs are the addresses of sectors which are accessed by the operating system, and PBAs are the addresses of flash pages (which consists of two parts, *i.e.*, the residing block number and the page number in the block). BL adopts a block-level translation mechanism for coarse-grained translation. An LBA under BL is divided into a virtual block address

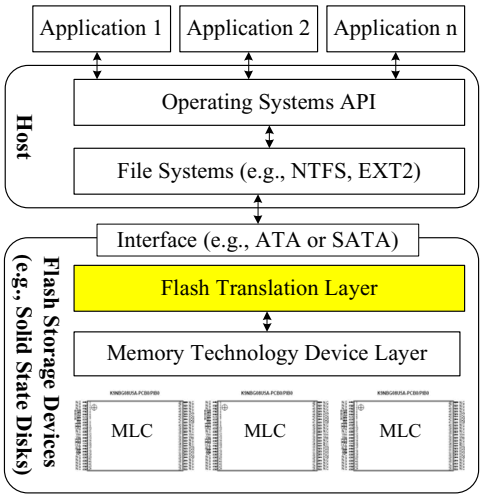


Fig. 1. System Architecture

(VBA) and a block offset (page number). Its translation table stores the mapping information from each given VBA to its PBA. The data is written to the PBA with the corresponding block offset. If the corresponding page is already used, a new block is allocated, and the valid data in the previous block are copied to the new block with the new written data. NFTL [24] also adopts a block-level address translation mechanism. The mapping mechanism is like BL, except that NFTL uses a replacement block to handle subsequent write requests. The contents of the (over-written) write requests are sequentially written to the replacement block. Therefore, an entry in the address translation table stores two physical block numbers, *i.e.*, primary block and replacement block. When a replacement block is full, the replacement block and the corresponding primary block are merged into a new primary block. The merge operation copies all of the valid data of the primary block and replacement block to a newly allocated primary block and erases the previous two blocks.

B. Motivation

Although NAND flash memory has been widely adopted in many storage-system designs due to various attractive features, its application domains quickly grow beyond its original targets. A well-known example could be solid state disks (SSDs). Note that SSDs are used to replace the role of hard disks. Because the accessing of user and system data often generates many small-sized writes, those small-sized writes could quickly deteriorate the write performance of SSDs, due to the out-place-update characteristics of flash memory and its slow erase operations.

As shown in Table I, in order to create a 64KB file in ext2, we need 11 write requests to different places, and only one of them is about the writing of some file contents. Most requests, except the request to the file contents, only access 8 LBAs per time. This observation implies harsh challenges on the designs of flash-memory management schemes. Although FTL has good performance on random writes of small sizes, its page-level translation mechanism needs a huge translation table, which is often too large to fit in the main memory. In order to resolve the size problem of translation tables, block-level translation mechanisms, *e.g.*, NFTL and BL, are invented for

large-scaled flash storage devices, *e.g.*, SSDs. However, they introduce considerable overheads of live-page copyings such that the system performance might be significantly affected.

File System	ext2
Number of total write requests	11
Number of total write requests to metadata	10
Number of write requests to the file content	1
Number of accessed LBAs	196
Number of accessed LBAs to metadata	68
Number of accessed LBAs to the file content	128

TABLE I
CREATING OF A 64KB FILE IN FILE SYSTEMS

MLC flash memory, instead of SLC flash memory, is usually selected as the storage media for low-cost embedded-system solutions. However, pages of MLC flash memory can only be written sequentially in a block, and partial-page programming is prohibited[4]. Such a limitation introduces extra overheads to writes over flash memory, and it might even make many existing management schemes infeasible. For instance, FTL and NFTL need to invalidate a page by updating the status bit of the spare area of a page, but the above constraints of MLC flash memory makes it impossible. Meanwhile, it is of paramount importance to maintain the sanity of file systems because the endurance of MLC flash memory is much lower than that of SLC flash memory, and embedded systems, especially portable devices, tend to run out of the battery during the run time. This research is motivated by the needs of performance and reliability enhancement over I/O-intensive applications with the considerations of physical constraints of MLC flash memory. Our goal is to propose a FTL (driver) design that better fits file systems, so as to reduce the overheads introduced by file systems and to improve the sanity of file systems after power losses. The technical problem is how to explore the attributes of file systems and their access patterns with the considerations of write constraints and the low reliability/performance problem of MLC flash memory. We should also explore the possibility in minimizing the number of live-page copyings caused by flash-memory management schemes.

III. A FILE-SYSTEM-AWARE FTL DESIGN

A. Overview

In order to improve the performance and reliability of file systems over MLC flash-memory storage systems, we propose a file-system-aware FTL (driver) design, where the characteristics of file systems are considered. In a file system, a systematic way is provided to organize data, and it usually consists of two parts: *metadata* and *userdata*. Metadata contain the general information of the file system, *e.g.*, the super-block and inode tables in ext2 , and the attributes of files and directories. Some modern journaling file systems might include journaling techniques to protect their metadata. Although metadata usually take a small part of the space of a storage system, it is frequently accessed with small sizes and therefore needs to be managed carefully. On the other hand, userdata stores the contents of files or user data. Compared to metadata, most userdata are less frequently accessed, and each access to userdata often touches data of a larger size.

[4]A 2KB page in SLC flash memory can partially be programmed for four times.

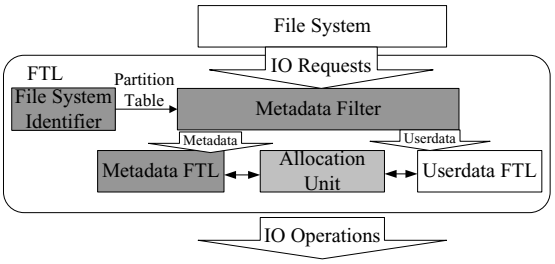


Fig. 2. The System Modules of the File-System-Aware Management Scheme

As shown in Figure 2, the proposed FTL design consists of five components: *file system identifier*, *Metadata filter*, *Metadata FTL*, *Userdata FTL*, and *Allocation unit*. The file system identifier is to identify the location of the file-system's metadata in the flash storage device by analyzing the partition table during the system's start-up. As a result, whenever the metadata filter receives an I/O request, it can distinguish metadata from userdata according to the accessed LBAs. If the accessed LBAs are belonging to userdata, they are passed to the userdata FTL where the userdata FTL adopts existing flash-memory management schemes, such as FTL, NFTL, and BL. If the accessed LBAs are in the metadata area, the metadata FTL is invoked to manipulate the underlying flash memory. The metadata FTL adopts the proposed "MFTL" management scheme that adopts a fine-grained address translation mechanism, so that the overhead of address translation and the number live-data copyings are reduced (Please see Section III-B). The MFTL requests free space in the unit of a block set, where a block set is a fixed number of blocks. In order to identify the version of stored data and to improve the performance on address translation, the last page of each block set stores the version number and the mapping information of the block set (Please see Section III-C). In addition, the stored version number and the mapping information can also help to recover the lost metadata of file systems so that the reliability and sanity of file systems can be improved as well. The allocation unit is to manage free space. Whenever metadata FTL or userdata FTL requests any free space, the allocation unit returns with the number of requested blocks. Once there are not enough free blocks, the allocation unit initiates the garbage collection to reclaim free space.

B. MFTL: A Metadata Management Scheme

In this section, we will first illustrate the general information and layout of a file system because the design of a file system has a significant impact on the performance of a storage system. We shall then provide some observations based on the access patterns of file systems and propose our approach in the handling of metadata. In this work, we propose to manage userdata with an existing flash-memory management scheme, such as NFTL, and adopt our metadata management scheme (MFTL) for metadata management.

MFTL, which is implemented in the metadata FTL (as shown in Figure 2), is specifically designed to manage the metadata of file systems efficiently. Note that different file systems do manage their general information in different ways. For example, ext2 divides the entire file system into several block groups and uses two metadata structures, *i.e.* Superblock and Group Descriptor Table (GDT), to describe the general information of the file system (as shown in Figure 3). The Superblock contains the basic information of the file system,

such as the block size, the number of blocks, and the number of inodes per block group. The GDT is to manage the block group information, and each entry of the GDT maintains the information of one block group, *e.g.*, the starting address of each block group and the status of blocks and inodes. Therefore, the layout of ext2 can be determined by scanning its Superblock and GDT. Each block group has its own "inode bitmap" and "block bitmap". Each bit in this bitmap indicates the allocation status of one data cluster or inode. Each block group also maintains an inode table, and each inode of the inode table describes the attributes of one file or directory in the group.

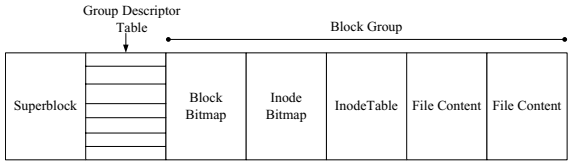


Fig. 3. The File Systems Layout

	Requests (R/W)	LBA	Sectors	Data Type
Metadata Requests	R	8	8	Group Descriptor Table
	R	1976	8	Inode Bitmap
	W	2	2	Superblock
	W	8	8	Group Descriptor Table
	W	1976	8	Inode Bitmap
	W	1984	8	Inode Table
	R	1968	8	Block Bitmap
	W	137296	128	File Content
Metadata Requests	W	6032	8	Directory
	W	2	2	Superblock
	W	8	8	Group Descriptor Table
	W	1984	8	Inode Table
	W	1968	8	Block Bitmap

Fig. 4. The Traces of Creating a 64KB File in ext2

Before any access to a file or directory, the file system might access related metadata to determine the system layout. The file system then accesses the metadata that contain the attributes of the accessed file or directory. If the file system needs to allocate or de-allocate any data clusters, the file system reads the corresponding bitmap structure to update the allocation status of the data clusters. Finally, the file system accesses the user data, *i.e., the contents of the file or directory*, and then updates the corresponding metadata if needed. Figure 4 shows the access patterns in the creating of a 64KB file in ext2. In order to determine the file system layout, the Superblock and Group Descriptor Table in ext2 are accessed. In order to allocate data clusters to a newly created file, the block bitmap in ext2 are accessed. In addition, the corresponding file attributes must be inserted to the inode table in ext2. Since the file system status has been changed, the updated information is written to the corresponding metadata to maintain the consistency of file system. It is observed that the traces contain many write requests to Superblock and Group Descriptor table.

The above observations show that metadata of a modern file system are frequently accessed and are of small sizes. The way to maintain the metadata should be different from user data so as to improve the performance of the file system, due to the out-place update characteristics of flash memory.

The metadata filter of the proposed FTL (driver) design is to identify I/O requests as *metadata requests* and *userdata requests*, where a metadata(/userdata) request is to access data belonging to metadata(/userdata). The metadata filter maintains a run-length structure for each metadata file in the main-memory. As shown in Figure 5, each I/O request contains information like S_LBA and $Length$, where S_LBA is the first LBA being accessed by the request, and $Length$ is the number of consecutive LBA's to be accessed. Based on the information, the metadata filter determine which of metadata or userdata FTL will handle the request. If the request is to access the content of a file or directory, it is sent to the userdata FTL, that adopts an existing flash-memory management scheme, such as NFTL or BL.

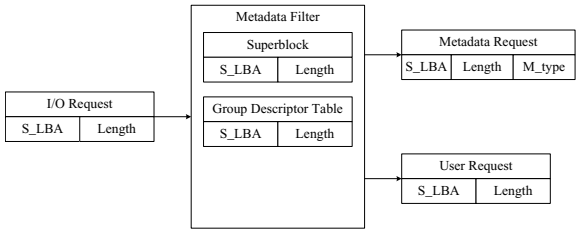


Fig. 5. Metadata Filter

MFTL, that is implemented in the metadata FTL, adopts a page-level address translation mechanism in a logging-similar fashion. The page-level address translation mechanism maintains one mapping entry for each page so as to reduce the address translation overhead and live-data copyings. Since metadata only occupy a small portion of file system, the memory requirement of the mapping table should be able to fit in most embedded systems MFTL writes data to a block from its first page one by one in a sequential way, *i.e.*, a logging-similar fashion. When there is no free space, MFTL requests for another block set to store new or updated data, where a block set is composed of a fixed number of blocks (Please see Section III-C). Metadata of different requests might be written to flash-memory pages in an interleaving way. The type of metadata (*i.e.*, M_type as shown in Figure 5) is stored in the spare area of each page, so that the type of data stored in a flash-memory page can be identified during the construction of address translation tables. When a block set is about to fill up, the last page, referred to as a *summary page*, is written with a version number and the page-level translation information of the metadata stored in the block set. Therefore, the address-translation overhead and the number of live-data copyings for frequently updated metadata can be reduced. The writing of a summary page is also referred to as a "commit" action to the writes of the corresponding block set.

C. Reliability versus Commit Actions

Block management should take the reliability of file systems into considerations, especially for battery-driven embedded systems. If a system crashes while data are written to a storage device, the file system in the storage device might be in an inconsistent state. In order to avoid the scanning of the entire file system during the recovery, some journaling technique is needed to protect sensitive data, such as metadata, where a journaling technique usually keeps update information in

"safe" area for recovery purpose. For example, Ext3, *i.e.*, a journaling version of ext2, stores the journaling information in some specific area in the unit of a *transaction*, which consists of a descriptor, updated metadata, and a committing status. Although the journaling technique is adopted by some file system to improve the integrity of a file system, it does introduce a significant amount of overheads, mostly in terms of writes.

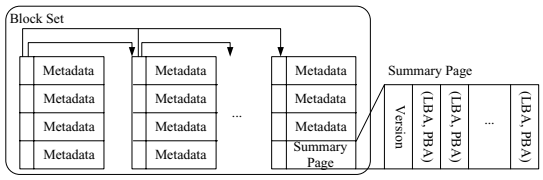


Fig. 6. A Block Set and its Summary Page

The design of MFTL not only aims at the system performance improvement but also introduces the idea of commit action for reliability considerations. Consider popular file systems that do not support the journaling technique, such as FAT32 and ext2. With MFTL, writes to each block set commit with a summary page for crash recovery, where MFTL allocates free space in the unit of a block set. As shown in Figure 6, inter-block link information is also maintained in the spare area of the first page of each block to speed up the block traversal in a set, where two links are used to point to its next block and the last block of the set, respectively. Pages of each block set are used to store metadata, except the summary page. Each summary page contains a 64-bit *version number* to indicate the version of the block set and the *page-level translation information* of the metadata in the block set. Whenever the summary page of a block set is written, the block set is said being "committed". When the file system crashes, MFTL will scan the summary page of each block set to recover the translation information of metadata according to their version numbers. If the system finds an uncommitted block set, then every page in the block set must be scanned. Any partial programmed page in an uncommitted block set should be discarded because the system might crash during some data writing. For metadata stored in discarded pages, their most recent versions in committed block sets should be used, where metadata in a committed block set are guaranteed being correct. When a bad block occurs, the system should also adopt the most recent version of metadata in a committed block set. The commit action of a block set can help in the reliability improvement of file systems with or without the support of the journaling technique because their journaling information might have a chance to get lost. The protection of metadata is improved by the commit action. Note that the block-set size could have an impact on the space utilization and might affect the worst-case amount of lost data after the system crash. That is because the last page of each block set is not used to store data, and parts of the data of an uncommitted block set could be discarded. The selection of the block-set size is a trade-off between the space utilization and the degree of reliability.

When multi-chip and multi-channel designs are considered for better performance, the proposed management scheme do have good potential in parallelism. Each block of a block set can reside in a different flash-memory chip. Since metadata are circularly written to blocks of a block set in a "z"-shaped scheme from their first pages, and pages in a block are sequentially written, the transmission time and programming time of metadata in a block set can be significantly reduced by writing multiple pages simultaneously. The read performance can be also improved in a similar way.

IV. PERFORMANCE EVALUATION

A. Performance Metrics and Experiment Setup

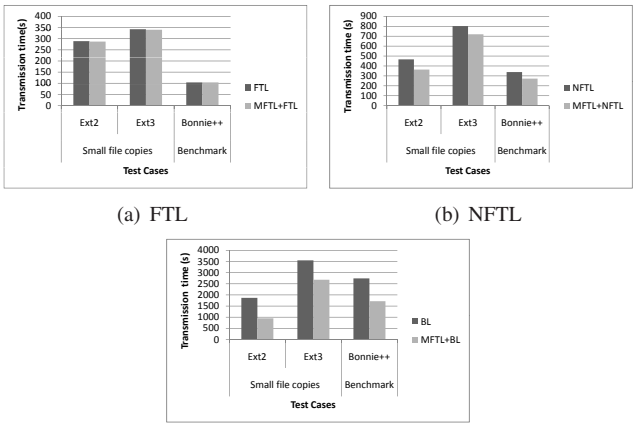
The purpose of this section is to evaluate the capability of the proposed FTL (driver) design with FTL, NFTL, and BL (implemented in the userdata FTL) over ext2 and ext3 file sysetms, in terms of performance improvement. The performance improvement was based on the data transmission time and the amount of live-data copyings.

The proposed FTL design was implemented on the *DM644X DaVinci* evaluation board to evaluate its performance under ext2 and ext3 file systems. The DaVinci evaluation board has 64MB on-board NAND SLC NAND flash memory (32 pages per block and 512 bytes per page). The proposed FTL design was implemented as a Linux device driver to control the on-board flash memory. 50MB of the on-board flash memory was partitioned as an ext2 or ext3 file system and each block set (allocated to store metadata) consisted of two blocks. The capability of the proposed FTL design was evaluated over some realistic cases and popular benchmarks. The realistic cases were the real archive (linux-kernel header files) to evaluate the performance of the proposed FTL design over ext2 and ext3. Meanwhile, the benchmark "Bonnie++" was used to further evaluate the performance over ext2 based on the industry practice.

B. Performance Improvement

1) Data Transmission Time: Figure 7 shows the performance improvement of the proposed FTL design with FTL, NFTL, and BL, compared with pure FTL, NFTL, and BL management schemes. For example, the improvement ratios on NFTL with small-file copyings and *Bonnie++* benchmark over the ext2 file system were 22% and 19%, respectively. Compared with the pure BL management scheme, which is widely used in USB flash drives, the performance improvement was more than 30%. It was worth of mentioning that there was no significant improvement, compared with the pure FTL management scheme, because the FTL management scheme adopted a fine-grained address translation mechanism for both userdata and metadata. However, the FTL management scheme is not practical in large-scale flash memory because it needs large RAM space to maintain the address translation table.

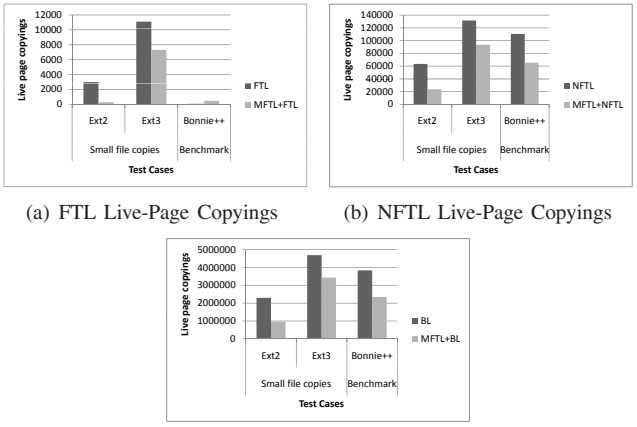
2) Live-Page Copyings: Figure 8 shows the number of live-page copyings with different management schemes over ext2 and ext3. The proposed FTL design could reduce more than 30% of live-page copyings in most cases. However, the number of live-page copings of the proposed FTL design was higher than that of pure FTL management scheme under the testing of the *Bonnie++* benchmark. This was because the data accessed by Bonnie++ were relatively small, and most of them were metadata. Therefore, the proposed FTL design had to do garbage collection many times to release free pages for metadata and resulted in many live-page copings in metadata. In contrast, the pure FTL management scheme could store metadata in any block of flash memory, and the amount of accessed userdata was relatively small so that the garbage collection was not triggered.



(a) FTL (b) NFTL

(c) BL

Fig. 7. Comparisons on the Transmission Time with Different Management Schemes



(a) FTL Live-Page Copyings (b) NFTL Live-Page Copyings

(c) BL Live-Page Copyings

Fig. 8. Comparisons on the Number of Live-Page Copyings with Different Management Schemes

V. CONCLUSION AND FUTURE WORK

This research is motivated by the significant performance degradation of file systems, such as ext2 and ext3, when flash memory is used as a storage medium. An efficient file-system-aware FTL design is proposed to better manage the metadata of file systems for performance and reliability considerations. With careful system behavior analysis, it was observed that the access frequency and small size characteristics of metadata introduce considerable overheads, *e.g.*, live-page copyings, to flash-memory management. A metadata filter is proposed to separate metadata requests and userdata requests based on the knowledge of the layout of file systems, and metadata are managed with a fine-grained address translation mechanism and written to flash-memory pages in a logging-similar fashion for performance and reliability considerations. An effective recovery scheme is then proposed to improve the integrity of a file system. The effectiveness of the proposed scheme was analyzed with case studies over ext2/ext3 file systems. A series of experiments also shows that the performance of ext2/ext3 file systems could be significantly improved by more than 20%, in most cases, where only limited extra main-memory was needed.

For future research, we shall further explore the trade-off between the main-memory consumption and performance improvement, especially for low-cost products. Multi-channel system designs will also be exploited for huge-capacity flash-memory storage devices for better performance and lower cost.

REFERENCES

[1] *K9NBG08U5M 4G * 8 Bit NAND Flash Memory Data Sheet*, Samsung Electronics, 2005.

[2] *NAND08Gx3C2A 8Gbit Multi-level NAND Flash Memory*, STMicroelectronics, 2005.

[3] B. Carrier, *File System Forensic Analysis*. Addison Wesley Professional, 2005.

[4] D. Woodhouse, "JFFS: The Journalling Flash File System," in *Ottawa Linux Symposium*, 2001.

[5] L.-P. Chang and T.-W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002, pp. 187–196.

[6] L.-P. Chang, "On efficient wear-leveling for large-scale flash-memory storage systems." 22st ACM Symposium on Applied Computing (ACM SAC), March 2007.

[7] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Endurance Enhancement of Flash-Memory Storage Systems: An Efficient Static Wear Leveling Design," *the 44th ACM/IEEE Design Automation Conference (DAC)*, June 2007.

[8] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," in *Proceedings of the 1995 USENIX Technical Conference*, Jan 1995, pp. 155–164.

[9] C.-H. Wu and T.-W. Kuo, "An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems," in *IEEE/ACM 2006 International Conference on Computer-Aided Design (ICCAD)*, November 2006.

[10] J.-H. Lin, Y.-H. Chang, J.-W. Hsieh, T.-W. Kuo, and C.-C. Yang, "A NOR Emulation Strategy over NAND Flash Memory," *the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications(RTCSA 2007)*, August 2007.

[11] "Flash File System. US Patent 540,448," in *Intel Corporation*.

[12] L.-P. Chang and T.-W. Kuo, "An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems," *ACM Symposium on Applied Computing (SAC)*, pp. 862–868, Mar 2004.

[13] Y. Du, M. Cai, and J. Dong, "Adaptive Energy-aware Design of a Multi-bank Flash-memory Storage System," *Proceedings of the 11 IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, 2005.

[14] J.-W. Hsieh, T.-W. Kuo, P.-L. Wu, and Y.-C. Huang, "Energy-Efficient and Performance-Enhanced Disks Using Flash-Memory Cache," *proceedings of ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED 2007)*, pp. 334–339, 2007.

[15] L.-P. Chang, "Hybrid Solid-State Disks: Combining Heterogeneous NAND Flash in Large SSDs," *The 13th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2008.

[16] *KFW8G16Q2M-DEBx 512M x 16bit OneNAND Flash Memory Data Sheet*, Samsung Electronics, 09 2006.

[17] *OneNAND Features and Performance*, Samsung Electronics, 11 2005.

[18] Y. Joo, Y. Choi, C. Park, S. W. Chung, E.-Y. Chung, and N. Chang, "Demand paging for onenandtm flash execute-in-place." CODES+ISSS, October 2006.

[19] "Flash Cache Memory Puts Robson in the Middle," *Intel*.

[20] "Windows ReadyDrive and Hybrid Hard Disk Drives, http://www.microsoft.com/whdc/device/storage/hybrid.mspx," Microsoft, Tech. Rep., May 2006.

[21] "DaVinci Digital Media System-on-Chip - TMS320DM6446 http: //focus.ti.com/docs/prod/folders/print/tms320dm6446.html," Texas Instruments, Tech. Rep.

[22] "Understanding the Flash Translation Layer (FTL) Specification, http://developer.intel.com/," Intel Corporation, Tech. Rep., Dec 1998. [Online]. Available: http://developer.intel.com/

[23] "FTL Logger Exchanging Data with FTL Systems," Intel Corporation, Tech. Rep.

[24] "Flash-memory Translation Layer for NAND flash (NFTL)," *M-Systems*, 1998.