The Sixteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications

An Efficient FTL Design for Multi-Chipped Solid-State Drives

Yuan-Hao Chang
Department of Electronic Engineering
National Taipei University of Technology
Taipei 106, Taiwan, R.O.C.
johnsonchang@ntut.edu.tw

Wei-Lun Lu, Po-Chun Huang, Lue-Jane Lee, and Tei-Wei Kuo
Department of Computer Science and Information Engineering
National Taiwan University
Taipei 106, Taiwan, R.O.C.
{*r96018, b91048, r98024, ktw*}*@csie.ntu.edu.tw*

***Abstract*—Although solid-state drives seem being excellent alternatives to replace hard disks in mobile devices, serious challenges arise due to performance and reliability concerns. This work targets performance enhancement designs with the considerations of low-cost MLC flash memory. In particular, an efficient flash management design is proposed to manage multi-chipped flash memory with cache support, where a two-level address translation mechanism is presented with an adaptive caching policy. The capability of the proposed approach is evaluated with a SystemC-based solid-state-drive simulator based on realistic workloads and benchmarks. It was shown that the proposed approach could significantly improve the performance of multi-chipped solid-state drives over various hardware configurations.**

***Keywords*-Flash memory; solid-state disk; performance; cache;**

I. Introduction

The applications of flash memory have grown beyond its original design. One popular example is solid-state drives (SSDs) for the replacement of hard drives. Due to the cost consideration, SSDs usually adopt low-cost multi-level-cell (MLC) flash memory as their storage media, although the performance of MLC flash memory is much worse than that of its counterpart single-level-cell (SLC) flash memory[1]. Such a development implies the potential popularity of multi-channel architectures with multiple chips for the designs of SSDs. As a result, many existing flash-translation-layer (FTL) designs are no longer efficient due to the lack of target architectural considerations and also the consideration of file access patterns. Such observations motivate this research on the design of an efficient FTL design that are more suitable to SSDs.

A NAND flash memory chip is composed of planes, each of which consists of blocks. Each block is of a fixed number of pages, and is the unit for erase operations. A page that is the basic unit for read/write operations contains a user area and a spare area, where the user area is for data storage, and the spare area stores the house-keeping information such as error detection/correction codes, status flags, and logical block addresses (LBAs). When a page is written, it can not be overwritten until its residing block is erased. This is called the *write-once property*. As a result, "out-place update" is usually adopted to write the to-be-updated data to another free page of flash memory due to the performance consideration. Pages with the latest copy of data are considered as live pages, and those with old data versions are dead pages. Thus, "address translation" is needed to map LBAs of data to their physical block addresses, and "garbage collection" is needed to reclaim invalid pages when there is not enough free space. In the past decade, researchers have proposed excellent flash-memory management designs, e.g. [1], [2], [3], [4], [5], [6], [7]. Some explored different system architectures and layer designs, e.g., [8], [9], and some exploited large-scaled storage systems and data compression, e.g., [10], [11]. Researchers also proposed various methods to improve the performance and reliability of native file systems over raw flash-memory media [12], [13], [14], [15], [16]. Some also considered how to improve the performance and reliability of NAND flash storage devices with non-volatile RAM such as phase-change memory [17], [18], [19].

In the past few years, vendors and researchers started exploiting the possibility to replace hard disks with solid-state drives in various product designs. In that direction, some researchers proposed adaptive address translation mechanisms to improve the performance on address translation and the handling of random write operations [20], [21]. Some proposed to adopt a write buffer in flash storage devices to improve the write performance by reducing the write frequency of data to flash-memory chips [22], [23], [24]. Although some presented adaptive striping architectures to enhance the degree of parallelism by accessing multiple planes/chips simultaneously, little work is done in the considerations of multi-channel architectures and the characteristics of file access patterns.

This work is motivated by the needs of performance enhancement for solid-state drives. We are interested in MLC-based flash-memory devices that include multiple chips distributed over multi-channels with new MLC write constraints (Please see Section II). Different from the previous work, a caching-oriented FTL design is proposed to manage multi-chipped flash storage devices with the consideration of the multi-channel and caching support. In particular, a two-level

[1]An MLC$_{\times n}$ cell can store n-bit information and an SLC cell can contain 1-bit information.

1533-2306/10 $26.00 © 2010 IEEE
DOI 10.1109/RTCSA.2010.37

address translation mechanism with the caching-oriented mapping structure is proposed to accelerate the address translation process for the data stored in either the cache or the flash memory. Note that such a translation mechanism is also applicable to other popular management designs, such as NFTL [1], [2], [3]. A chain-based block management with an adaptive caching policy is proposed to enhance the read/write performance of multi-channel flash storage devices. The proposed block management could support fast crash recovery and efficient system initialization without the support of the version number technique. The proposed design is evaluated with a SystemC-based solid-state-drive simulator. A series of experiments was conducted based on realistic workloads and representative benchmarks. We show that the proposed design could significantly improve the performance of solid-state drives over various hardware configurations, compared to existing popular FTL designs.

The rest of this paper is organized as follows: Section II presents the system architecture and the motivation of this work. In Section III, a caching-oriented FTL is proposed. Section IV summarizes the experiment results on the performance enhancement. Section V is the conclusion.

II. System Architecture and Research Motivation

A. System Architecture

A NAND flash-memory chip might consist of multiple subchips, and each subchip could operate independently. A subchip might contain multiple planes, and some chips support the two-plane operation to access two adjacent planes of the same subchip in parallel so as to enhance the degree of parallelism. Each plane is partitioned into blocks and a block is further divided into a fixed number of pages, where a block and a page are the units of erase and read/write operations, respectively. In an SLC flash memory, each block might consist of 64 pages, and each page can store 2KB data and 64B spare information [25]. In comparison, each block of MLC$_{\times 2}$ flash memory might have 128 pages, each of which can store 2KB(/4KB) data and 64B(/128B) spare information [26], [27]. Due to the cost considerations, flash-memory storage systems usually adopt MLC flash memory, but it imposes two new write constraints. That is, pages of a block must be sequentially written from the first one, and it can not be partially programmed/written. As a result, the existing FTL designs that needs to update status flags to invalidate a page or to write pages of a block in a non-sequential order become infeasible or lack of efficiency.

A typical multi-chipped NAND flash-memory storage system, such as a solid-state drive (SSD), is usually composed of multiple chips. One or more chips share the same data bus to form a *channel*, and all of the channels can do data transferring simultaneously. As shown in Figure 1, a multi-chipped flash-memory storage system is usually connected to a host system through a standard host interface. e.g, Serial

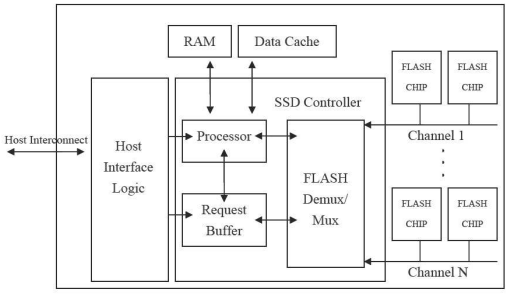

Figure 1. The Block Diagram of a Multi-Chipped Flash-Memory Storage System

Advanced Technology Attachment (SATA). A controller of a flash-memory storage system, that controls flash media, consists of three major components: the *processor*, the *request buffer*, and the *multiplexer*. The flash translation layer (FTL) firmware is executed on the processor to support address translation, garbage collection, and wear-leveling. Address translation maps any given logical block address (LBA) to its corresponding physical block address (PBA) on the flash memory. Garbage collection reclaims the space occupied by out-of-date data, whenever needed, because any updating of data on a page must be written to a free page unless its residing block is erased. Wear-leveling is to distribute block erases as evenly as possible over a flash-memory chip so as to lengthen its lifetime. The request buffer is a buffer that stores the data to be transferred between the host system and flash-memory chips. The multiplexer receives commands and transfers data to/from a selected chip according to the physical addresses of a received command. In addition to the controller, there are two additional memory devices, i.e., the RAM and the data cache, controlled by the processor. The RAM maintains the data structures and address translation information for the management of flash-memory chips, and the data cache is used to cache data according to the FTL design so as to improve the read/write performance of the storage system.

B. Research Motivation

As the density and capacity of flash memory chips keep increasing, the performance of flash memory becomes worse. As shown in Table I, the time of a read/write operation to a MLC$_{\times 2}$ page is much longer than that to a SLC page. This problem is exaggerated when MLC-based storage devices (e.g., SSDs) are adopted in applications with high access frequencies such as a major secondary storage device of computer systems (for the replacement of hard drives). Although many flash-memory storage systems include multi-channels with multiple chips to improve the storage performance by the enhancement of the degree of parallelism, most of the existing FTL designs do not fully utilize the bus bandwidth and the parallelism of flash chips.

For instance, two-planes of the same subchip could be accessed simultaneously. Subchips of a chip and chips of the same bus/channel could also operate independently, and they could be accessed in an alternative way. Furthermore, chips of different buses or channels could be accessed in parallel. Such facts should be considered in the management strategy and space allocation of an FTL design for flash-memory storage systems like SSD's. Moreover, most of the existing FTL designs, such as well-known NFTL [28], [1], [29], [2], often invalidate a page by updating the status bit in the space area of a page, and they assume that data can be written to pages randomly within a block. Such assumptions and designs are no longer possible because of new constraints on the write operations of MLC flash memory, due to its unique characteristics.

Type & Part Number	SLC	$MLC_{\times 2}$
Price (US dollars/GB, 2009 Q1)[30]	5.65	1.57
Serial Access	$25ns$	$25ns$
Random Read	$20\mu s$	$60\mu s$
Write/Program	$200\mu s$	$800\mu s$
Erase	$1.5ms$	$1.5ms$
Partial Page Write/Program	Yes (4)	No (1)
Random Page Write	Yes	No

Table I
THE CHARACTERISTICS OF SLC AND MLC FLASH MEMORY [25], [26]

This research is motivated by the demands of the performance enhancement of multi-chipped flash-memory storage systems such as SSD's. We shall consider the potential system architecture of SSD's and the locality of access patterns, where access locality could play a major role in performance improvement when user behaviors are considered. We must point out that the address translation consideration of FTL designs could be further utilized to support caching designs to address the locality of access patterns so that a joint design with less overheads is possible. New write constraints imposed by MLC flash shall also be considered. One major technical issue is how to explore the parallelism of multiple chips over a multi-channel architecture with the considerations of bus utilization and space allocation of flash-memory chips. A caching design should be integrated in the mapping information maintenance of FTL designs. At the same time, management overheads including the crash recovery time and system initialization time must be under control with the fast-growing capacity of SSDs.

III. A MULTI-CHIPPED FTL WITH CACHING SUPPORT

A. Overview

The purpose of this section is to present an FTL design for multi-chipped SSD's. An address translation scheme with caching support is proposed so as to explore a high degree of parallelism for flash-memory access and to reduce overheads on the management of address translation. As

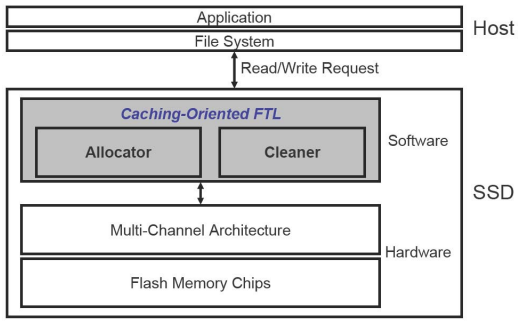

Figure 2. Components of Caching-Oriented FTL and Hardware

shown in Figure 2, the proposed FTL design (referred to as Caching-Oriented FTL) consists of two components: the *allocator* and the *cleaner*. The allocator manages the space of flash memory and data cache with a two-level address translation scheme and an adaptive caching policy (Section III-B), where the cleaner is to reclaim the space of selected invalid data when there are not enough free pages (Section III-C3). The two-level address translation scheme maintains an address translation table to translate any given LBA to their corresponding PBA or cache slot, depending on whether the corresponding data are cached. The adaptive caching policy allocates the cache slot in the unit of multiple pages (referred to as a page set), such that data could be read from or written to multiple flash-memory chips simultaneously, where the size of cache slots is predetermined based on the number of channels and the average number of accessed LBAs of each request. An adaptive caching policy with a heap-like index structure will be presented to locate the slot for cache replacement.

The cleaner is to reclaim occupied space by merging physical block sets that are mapped to the same virtual block, where a physical block set contains a fixed number of physical blocks, and each virtual block is of a fixed number of virtual pages, each of which contains a fixed number of consecutive LBAs. Note that the size of one virtual page equals to that of one cache slot. With the consideration of the multi-channel architecture, physical blocks of the same physical block set are allocated at chips of different channels, and physical pages with the same offset to their corresponding block set form a physical page set. Whenever a physical block set of a virtual block is filled up, another physical block set is allocated for the virtual block, and all of the physical block sets mapped to the same virtual block are chained together. The cleaner maintains a bit array for each virtual block to know whether a virtual block is mapped to more than one physical block set. As a result, victim physical block sets could be efficiently detected by scanning their corresponding bit array.

B. A Caching-Oriented Mapping Mechanism

1) Address Translation with Caching Support: In this section, an address translation mechanism with caching support is proposed to enhance the performance of address translation and to reduce the required RAM space in the maintenance of translation information. The mechanism translates given LBAs to their corresponding PBAs in the unit of one virtual page, where the number of LBAs of a virtual page equals to the number of sectors that can be stored in a physical page set. Each virtual page of a virtual block might correspond to any physical page set of the physical block sets that are mapped to this virtual block. Note that a physical page(/block) set is the unit of read-write(/erase) operations.

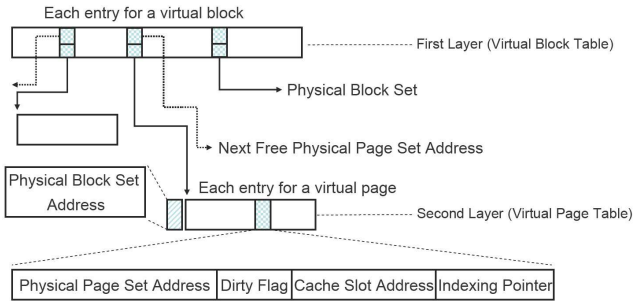

Figure 3. The Two-Level Address Translation Scheme

As shown in Figure 3, a two-level address translation mechanism is adopted. The virtual block address (VBA) of an LBA is obtained by dividing the LBA with the number of LBAs per virtual block, and the virtual page address (VPA) of the LBA is the quotient in the division of the remainder of the former division by the number of LBAs per virtual page. Given an LBA, its VBA serves as an index to look up the virtual block table for its corresponding virtual page table. Its VPA is then used to look up the corresponding virtual page table for the physical page set that contains the data of the given LBA. The virtual block table is written back to flash memory during the system's shutdown and is loaded to RAM during the system's start-up. Each entry of the table maintains two pointers: One points to a free physical page set for the corresponding virtual block so as to avoid the scanning of physical block sets for free space, and the other points to its corresponding virtual page table that might be cached in RAM or stored in flash memory.

Virtual page tables could be cached in RAM and replaced/written back to flash memory in the least-recently-used (LRU) fashion based on their referenced time. Each entry of a virtual page table maintains the mapping information of the virtual page: The address of the corresponding physical page set, a cache slot address that indicates whether the data of the corresponding virtual page is cached and where it is if it is cached, and an indexing pointer that points to the corresponding node in the heap-like index structure. In the proposed mechanism, a heap-like index structure is used to locate the cache slot for cache replacement (Please see Section III-B2). When a virtual page table is cached, the physical address of the virtual page table should be saved in the cache as well because a cached virtual page table might be discarded before it is modified. In the meantime, when a cached virtual page table is to be replaced, it should be written back to flash memory if its corresponding dirty flag is set.

2) Heap-Like Index Structure: In order to improve the performance on the (data) cache replacement, a heap-like index structure is proposed. This index structure is composed of two min heaps. That is, a clean heap and a dirty heap to maintain the access information of the clean (or unmodified) and dirty (or modified) virtual pages that are cached, respectively. Note that a node at the root of a min-heap is the least-important one in the heap. Each node in the heaps is associated with one cached virtual page to maintain the number (i.e., cnt) of the referenced times since the virtual page was cached and the latest referenced time (i.e., t_l) of the virtual page. The heap value of each node is defined as $\frac{cnt}{t_n - t_l}$, such that the importance of a cached virtual page could be easily identified according to the access frequencies and the elapsed time after the virtual page was referenced last time, where t_n is the time when the node is examined. This is because a node that is referenced recently and frequently should be kept in the heap so as to enhance the cache hit ratio, but a node that was referenced frequently but has not been referenced recently should be removed from the cache due to the consideration of temporal locality.

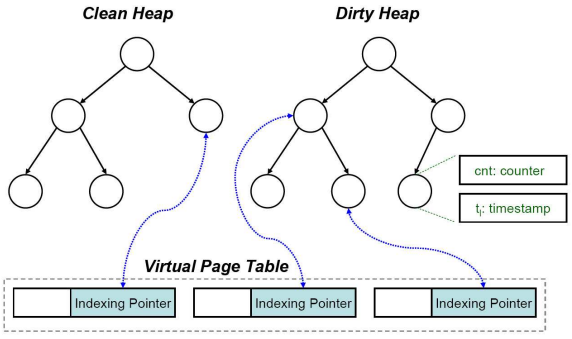

Figure 4. Heap-Like Index Structure

When a virtual page is referenced, the cnt and t_l of the corresponding node in the heap is advanced by one and updated to the referenced time, respectively. Then this node is moved upward or downward to maintain the min-heap property. When a cached clean virtual page is modified, its corresponding node in the clean heap should be removed from the clean heap and attached to the dirty heap. Then both of the clean heap and dirty heap are rearranged to maintain the full-binary-tree property and the min-heap property. Note

that the time complexity to maintain the min-heap property on the insertion or deletion of a node is $O(\lg n)$, where n is the number of nodes in the heap. If a read(/written) virtual page that is not in the cache and the cache is not full, the virtual page is loaded to the cache, and a node related to the virtual page is created and attached to the clean(/dirty) heap and moved upward to maintain the min-heap property. When the data cache is full and a virtual page needs to be cached, the virtual page corresponded to the root node of either the clean heap or the dirty heap should be replaced because it is either not recently or not frequently accessed. This can prevent a virtual page that was frequently accessed but is not recently accessed from staying in the cache, so that the cache hit ratio could be improved. In addition, the replacement of a dirty(/clean) virtual page introduces(/doesn't introduce) a write operation to write the modified data back to flash memory, but the replacement of a frequently read virtual page might introduce more cache misses. Therefore, the proposed adaptive caching policy should replace the virtual page corresponded to the root node of either the clean heap or the dirty heap with the considerations of the access frequencies of virtual pages, the overheads of a cache miss, and the overheads on the replacement of a clean or dirty virtual page (Please refer to Section III-C1).

C. Access Strategy

1) An Adaptive Caching Policy: In this section, an adaptive caching policy is proposed to improve the performance of the data cache. It is adaptive to different access patterns with the considerations of the access frequencies of virtual pages and the overheads on the replacement of clean or dirty virtual pages due to cache misses. When a read request is received, the two-layer address translation mechanism is looked up to find out whether the requested data are cached. If the requested data are cached, the data are simply returned and the corresponding nodes in the heaps are updated accordingly. If any request data are not cached, they are loaded from the flash memory in the unit of one virtual page. Meanwhile, if the size of the requested data is smaller than the size of one virtual page, they are cached with their corresponding nodes created in the clean heap and with their corresponding translation tables updated accordingly, because frequently accessed data are usually accessed randomly and inconsecutively [31], [32]. Note that the size of one virtual page is predetermined based on the number of channels and the average number of accessed LBAs of each request. When a write request is received and the accessed virtual pages are cached, the written data are simply written to the cache slots of the corresponding virtual pages. Then corresponding nodes of the virtual pages are updated accordingly, and are moved to the dirty heap if they are originally in the clean heap. If any accessed virtual page is not cached, and the size of the requested data is larger than that of one virtual page, the written data to this virtual page are simply written to the flash memory directly (since sequential accessed data are usually accessed infrequently); otherwise, the written data are cached, and the nodes of the corresponding virtual pages are created and placed in the proper positions of the dirty heap.

When the cache is full and the requested data of a read/write request needs to be cached, the caching policy replaces the virtual page pointed by the root node of either the clean heap or the dirty heap according to the heap values in the two root nodes. Since the replacement of a dirty virtual page imposes a write operation to write the virtual page back to the flash memory, and a read miss introduces a read operation to read the missed virtual page from the flash memory, the importance of a cached dirty virtual page is at least α times of that of a cached clean virtual page, where α could be the ratio of the time on the writing of a virtual page to the time on the reading of a virtual page. Therefore, when the heap value in the root node of the clean heap is smaller(/larger) than α times of that of the dirty heap, the virtual page corresponded to the root node of the clean(/dirty) heap is replaced. As a result, the number of cache slots allocated to cache dirty virtual pages and clean virtual pages could be adjusted automatically according to the read/write access patterns.

2) Chain-Based Block Management: As shown in Figure 5, a chain-based block management is proposed to manage flash memory in the unit of one physical block set so as to support the fast crash recovery and to enhance the performance on read/write requests and garbage collection (Please see III-C3) with the consideration of the MLC write constraints. In the proposed block management, each virtual block is mapped to at most $n_b + 1$ physical block sets, where n_b is the number of physical blocks in a physical block set. Among the physical block sets mapped to the same virtual block, the first one is the primary physical block set, and others are the replacement physical block sets. All of the replacement physical block sets of a virtual block points to their corresponding primary physical block set, where the pointers could be kept in the spare area of physical pages. Physical pages except the ones in the last two physical page sets of the primary physical block set are *data pages* used to store user data. In a primary physical block set, physical pages of the last but one physical page set are *summary pages* to store the virtual page table of the corresponding virtual block, and the physical pages of the last physical page set are *replacement pages* that point to the their corresponding replacement physical block sets. Note that the size of a virtual page equals to that of one physical page set, and the number of virtual pages in a virtual block equals to the number of physical page sets used to stored user data, so that the data of one virtual block could be fitted in one primary physical block set to enhance efficiency of garbage collection (Please see Section III-C3).

Initially, each virtual block is not mapped to any physical

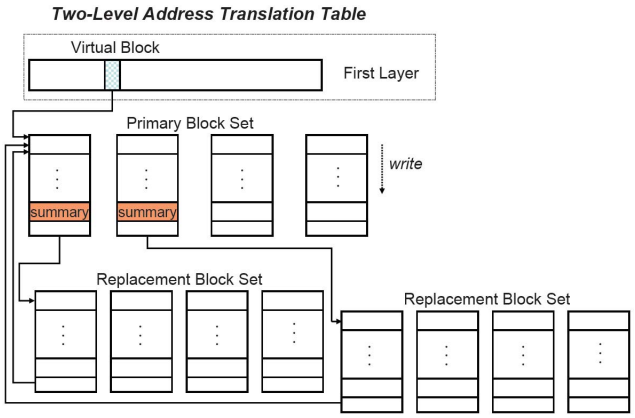

Figure 5. Chain-Based Block Management

block set. When data of a virtual block is written to the flash memory for the first time, a free physical block set is allocated and becomes its primary physical block set. Note that data are written to a physical block set sequentially from the first physical page set. If all of the data pages in the primary physical block set are filled up or the virtual page table of the virtual block is removed from the cache, the virtual page table of the virtual block is written to the summary page of the first physical block of the primary physical block set so as to "commit" the primary physical block set. Then another free physical block set is allocated as the first replacement physical block set of the virtual block to store the further written data, and the replacement page of the first physical block of the primary physical block set points to this newly allocated free physical block set. When the replacement block set is filled up or the virtual page table of the virtual block is removed from the cache, the virtual page table of the virtual block is written to the summary page of the second physical block of the primary physical block set, and the replacement page of the second physical block of the primary block set points a free physical set that is allocated as the second replacement physical block set of the virtual block. This procedure is repeated until the n_bth replacement physical block set is allocated. After the n_bth replacement physical block set of the virtual block is allocated, all of the valid data stored the $n_b + 1$ physical block sets of the virtual block are merged to a free physical block set when the n_bth replacement physical block set is filled up or the virtual page table of the virtual block is to be removed from the cache. Then the $n_b + 1$ old physical block sets are erased and returned back to the *free block list*, and this newly allocated physical block set becomes the new primary physical block set of the virtual block, where the free block list is a queue to maintain the list of free physical block sets that are erased and ready to be allocated for the data storage. Note that the proposed chain-based block management can reserve an swapping area to maintain the virtual page tables that are removed from the cache when their corresponding (primary or replacement) physical block set is not filled up, so that the space utilization of flash memory could be improved.

The proposed block management supports *crash recovery* over MLC flash memory without the support of the version number (that is to maintain a unique serial number for each write operation or each version of an LBA), because each virtual block is mapped to at most $n_b + 1$ physical block sets, and the allocation sequence of the replacement physical block sets of a virtual block could be derived by checking the replacement pages in the primary physical block set of the virtual block. Meanwhile, a virtual block could efficiently recover its corresponding virtual page table by scanning its last replacement physical block set after the system crashes. That is because each virtual block has at most one uncommitted physical block set that must be its last replacement physical block set. On the other head, the virtual block table could also be efficiently recovered because all of the replacement physical block sets of a virtual block point to their corresponding primary block set that keeps the virtual page table of the virtual block. The crash resistance of the virtual block table could be further improved by the dual buffer concept. The virtual block table and the free block list can be written to the dual buffer together as a *log version*, where the dual buffer is allocated in the flash memory, and the free block list always maintains at least a predetermined number (e.g., 256) of free physical block sets. Once the free physical block sets kept in the latest log version are used out, the virtual block table and the new free block list are written back to the dual buffer in the flash memory as a new log version. If the system crashes before the modified virtual block table is written to the flash as a new log version. The up-to-date virtual block table could be reconstructed by scanning the free physical block sets that are kept in the latest log version to update the missing information from the virtual block table in the latest log version. As a result, the time on the crash recovery is limited by the number of the free physical block sets in the latest log version, so that the translation information (or tables) could be fast recovered without scanning most physical block sets in the flash memory. Note that the dual buffer in the flash memory could be mirrored to strengthen the resistance to burst errors such as flash page/block failures.

3) Circular-Based Cleaner: In order to accelerate the performance on garbage collection with the consideration of wear leveling, a bit array is maintained to indicate the status of each virtual block. In this bit array, each bit is a flag to indicate the status of one virtual block. When a virtual block is mapped to more than one physical block set, its corresponding flag is set; otherwise, it is cleared. When the number of free physical block sets in the free block list is lower than the predetermined threshold, the cleaner is activated to scan the bit array in a circular chain fashion

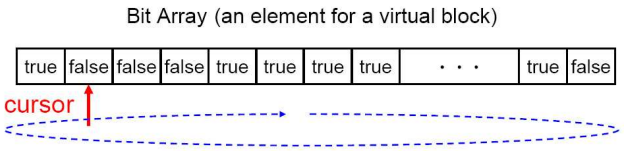

Figure 6. The Garbage Identifying Structure

until a set flag is found. Then the physical block sets mapped to the virtual block of the set flag are merged to a free physical block set that becomes the primary physical block set of the virtual block, and the existing physical block sets are erased and put into the end of the free block list. This procedure is repeated until the number of physical block sets in the free block list is larger than the predetermined threshold. Note that the circular scanning of virtual blocks in the selection of victim physical block sets is very effective in the implementation, and could release the replacement physical block sets of cold data so as to increase the number of physical block sets for hot data. The design is close to a random selection policy in reality because virtual block could virtually exist in any physical block set of the flash memory. Since the size of the valid data in a virtual block is no more than the size of the user data that could be stored in a primary physical block set, the cleaner can generate at least one additional free physical block set after the merging of the physical block sets of a virtual block. Thus, the worst-case time on free physical block set allocation can be bounded and estimated, and this property is critical to real-time applications. To achieve the wear-leveling efficiently and effectively, the cleaner could reclaim the physical block set of a virtual block whose corresponding flag is not set whenever a fixed number of block erases has been done [33], [34], [35], [36].

IV. Performance Evaluation

A. Performance Metrics and Experimental Setup

The purpose of this section is to evaluate the capability of the proposed caching-oriented FTL (referred to as *COFTL*), in terms of performance. The performance of the strategy was evaluated based on the throughput of the read/write requests per second, and also evaluated over various configurations. Different cache sizes and different numbers of channels were considered in this experiment. The proposed caching-oriented FTL was compared with the *BL* strategy because BL is widely adopted in various flash devices due to its simplicity and small RAM usage. In this experiment, BL adopts a block-level translation mechanism. Each LBA is divided into a virtual block address (VBA) and a block offset (page number), and is mapped to its PBA by a translation table.

As shown in Table II, 8GB(i.e.,64Gb) MLC flash-memory chips were under investigation, and 97% of the storage system was initially stored with data. The capability of the management strategies was evaluated with a SystemC-based solid-state-drive simulator through a realistic trace, where the SystemC-based simulator simulates the handling of read/write operations in the unit of $10ns$ with the considerations of the data transmission time and the data/address bus contention issues. Each channel is composed of one chip. The trace was collected over a desktop PC with a 160GB hard disk (by NTFS). The workload was mainly on daily activities, such as document editing, email accessing, file uploading and downloading, and web surfing.

Chip size	2 8GB (2 planes)
Plane size	4096 blocks (4GB)
Block size	256 pages (1024KB+56KB)
Page size	4KB + 224B
Page write/Programe	$900\mu s$
Page read	$50\mu s$
Block erase	$3ms$
Transfer time (per byte)	10 ns

Table II
CHARACTERISTICS OF THE EVALUATED MLC FLASH CHIP [37]

B. Experimental Results

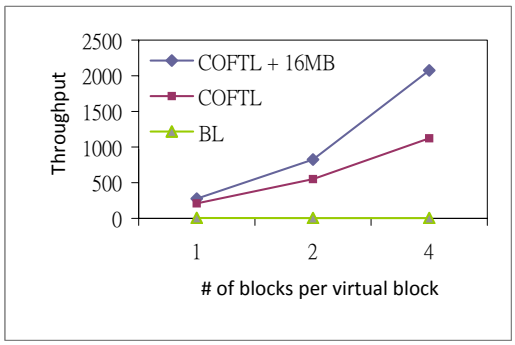

Figure 7. Throughput Comparison (16 Channels)

1) Access Performance: Figure 7 shows that the proposed COFTL could greatly improve the access performance of the flash-memory storage system, compared to BL, where the x-axis denotes the number of blocks per virtual block and the y-axis denotes the throughput (i.e., the average number of completed read/write requests per second). This was because COFTL adopted a two-level address translation scheme to optimize the parallel access to the flash chips through multi-channels. Meanwhile, when the number of channels in the solid-state drive was kept the same, the performance of COFTL was significantly improved as the number of blocks per virtual block increased. The reason was that COFTL utilized the hardware multi-channel to access each channel of the same virtual block concurrently.

2) Configuration Considerations: Figure 8 shows the throughput of the proposed COFTL with different cache sizes, where the x-axis denotes the cache size, and the y-axis denotes the throughput. When the cache size increased, the throughput increased as well. However, the performance of COFTL became saturated when the cache size was no less than 16MB.

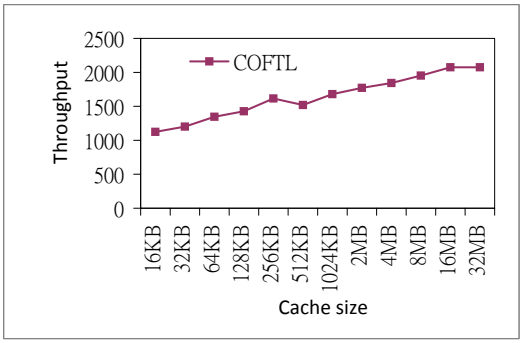

Figure 8. Throughput with Different Cache Sizes (4 Blocks per Virtual Block, 16 Channels)

In solid-state drives, the number of channels could greatly affect the performance of the proposed COFTL. When the number of blocks per virtual block was fixed, the performance of COFTL could have significant improvement due to the increased degree of parallelism (as shown in Figure 9). In other words, when a virtual block was being accessed, the subsequent requests to virtual blocks could be handled concurrently if their corresponding channels were idle.

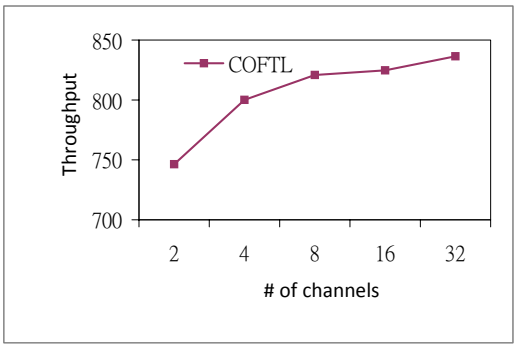

Figure 9. Throughput with Different Number of Channels (Cache Size: 16MB, 2 Blocks Per Virtual Block)

V. Conclusion

This work is motivated by the needs of performance enhancement for solid-state drives. In this paper, a caching-oriented FTL design is proposed to manage multi-chipped flash storage devices with the consideration of the multi-channel caching support, and MLC-flash write constraints. In particular, a two-level address translation mechanism with the adaptive caching-oriented mapping structure is proposed to accelerate the address translation process for the data stored in either the cache or the flash memory. A chain-based block management with an adaptive caching policy is proposed to enhance the read/write performance of multi-channel flash storage devices. The proposed block management could support fast crash recovery and efficient system initialization without the support of the version number technique. With a series of experiments over a SystemC-based solid-state-drive simulator, the proposed design could significantly improve the performance of solid-state drives over various hardware configurations, compared to existing popular FTL design.

In the future, we shall evaluate the performance of the proposed FTL design with more existing FTL designs. We shall also further exploit the possibility to enhance the degree of parallelism, and the possibility to cooperate with existing management schemes.

Acknowledgment

This work was supported in part by the NSC under grant Nos. 99-2218-E-027-005 and 98-2221-E-002-120-MY3, by the Excellent Research Projects of National Taiwan University under grant No. 98R0062-05, and by the ROC Ministry of Economic Affairs under grant No. 98-EC-17-A-01-S1-034 in Taiwan.

References

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

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

[3] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A Design for High-performance Flash Disks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 88–93, 2007.

[4] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Endurance enhancement of flash-memory storage, systems: An efficient static wear leveling design," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, June 2007, pp. 212–217.

[5] Y.-H. Chang and T.-W. Kuo, "A Commitment-based Management Strategy for the Performance and Reliability Enhancement of Flash-memory Storage Systems," in *the 46th ACM/IEEE Design Automation Conference (DAC)*, 2009.

[6] Y.-L. Tsai, J.-W. Hsieh, and T.-W. Kuo, "Configurable nand flash translation layer," in *Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on*, vol. 1, June 2006, pp. 8 pp.–.

[7] C.-H. Wu, T.-W. Kuo, and C.-L. Yang, "A space-efficient caching mechanism for flash-memory address translation," in *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, April 2006, pp. 8 pp.–.

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

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

[10] Q. Xin, E. L. Miller, T. Schwarz, D. D. Long, S. A. Brandt, and W. Litwin, "Reliability Mechanisms for Very Large Storage Systems," in *the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS)*, Apr 2003, pp. 146–156.

[11] K. S. Yim, H. Bahn, and K. Koh, "A Flash Compression Layer for SmartMedia Card Systems," *IEEE Transactions on Consumer Electronics*, vol. 50, no. 1, pp. 192–197, Feburary 2004.

[12] Y. J. Cho and J. W. Jeon, "Design of an efficient initialization method of a log-based file system with flash memory," in *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, July 2008, pp. 1620–1625.

[13] S. Kim and Y. Cho, "The design and implementation of flash cryptographic file system based on yaffs," in *Information Science and Security, 2008. ICISS. International Conference on*, Jan. 2008, pp. 62–65.

[14] C. Lee, S. H. Baek, and K. H. Park, "A hybrid flash file system based on nor and nand flash memories for embedded devices," *IEEE Trans. Comput.*, vol. 57, no. 7, pp. 1002–1008, 2008.

[15] S.-H. Lim and K.-H. Park, "An efficient nand flash file system for flash memory storage," *Computers, IEEE Transactions on*, vol. 55, no. 7, pp. 906–912, July 2006.

[16] Y. Park, S.-H. Lim, C. Lee, and K. H. Park, "Pffs: a scalable flash memory file system for the hybrid architecture of phase-change ram and nand flash," in *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 1498–1503.

[17] I. H. Doh, H. J. Lee, Y. J. Moon, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Impact of nvram write cache for file system metadata on i/o performance in embedded systems," in *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, 2009, pp. 1658– pages = 1658–1663, location = Honolulu, Hawaii, doi = http://doi.acm.org/10.1145/1 529 282.1 529 654, address = New York, NY, USA,.

[18] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha, "Performance trade-offs in using nvram write buffer for flash memory-based storage devices," *Computers, IEEE Transactions on*, vol. 58, no. 6, pp. 744–758, June 2009.

[19] S. Park, H. Jung, H. Shim, S. Kang, and J. Cha, "Using non-volatile ram as a write buffer for nand flash memory-based storage devices," in *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, Sept. 2008, pp. 1–3.

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

[21] C.-H. Wu and T.-W. Kuo, "An Adaptive Two-level Mnagement for the Flash Translation Layer in Embedded Systems," in *the IEEE/ACM Iinternational Conference on Computer-Aided Design (ICCAD)*, 2006, pp. 601–606.

[22] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "FAB: Flash-aware Buffer Management Policy for Portable Media Players," in *IEEE Transactions on Consumer Electronics*. IEEE, 2006, pp. 485–493.

[23] H. Kim and S. Ahn, "BPLRU : A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 239–252.

[24] S. yeong Park, D. Jung, J. uk Kang, J. soo Kim, and J. Lee, "CFLRU: a Replacement Algorithm for Flash Memory," in *the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2006.

[25] *K9K8G08U0M 1G * 8 Bit NAND Flash Memory Data Sheet*, Samsung Electronics, 2005.

[26] *K9GAG08U0M 2G x 8bit NAND Flash Memory Data Sheet*, Samsung Electronics, September 2006.

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

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

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

[30] *Flash Contract Price,http://www.dramexchange.com/*, DRAMeXchange, 03 2009.

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

[32] P.-L. Wu, Y.-H. Chang, and T.-W. Kuo, "A File-System-Aware FTL Design for Flash-Memory Storage Systems," in *the ACM/IEEE Design, Automation and Test in Europe (DATE)*, 2009.

[33] A. Ban, "Wear Leveling of Static Areas in Flash Memory. US Patent 6,732,221," *M-systems*, 2004.

[34] A. Ben-Aroya and S. Toledo, "Competitive Analysis of Flash-memory Algorithms," in *the 14th Conference on Annual European Symposium (ESA)*, 2006, pp. 100–111.

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

[36] M. Spivak and S. Toledo, “Storing a Persistent Transactional Object Heap on Flash Memory,” in *the 2006 ACM Conference on Language, Compilers, and Tool Support for Embedded Systems (LCTES)*, 2006, pp. 22–33.

[37] *MT29F64G08CFABB 64Gb NAND Flash Memory Data Sheet*, Micron, 2008.