Configurable Flash-Memory Management: Performance versus Overheads

Jen-Wei Hsieh, *Member*, *IEEE*, Yi-Lin Tsai, Tei-Wei Kuo, *Senior Member*, *IEEE*, and Tzao-Lin Lee

Abstract—Flash memory has been widely adopted in various consumer products for information storage, especially for embedded systems. With strong demands on product designs for overhead control and performance requirements, vendors must have an effective design for the mapping of logical block addresses (LBAs) and physical addresses of data over flash memory. This paper targets such an essential issue by proposing a configurable mapping method that could trade the main-memory overhead with the system performance under the best needs of vendors. A series of experiments is conducted to provide insights on different configurations and the proposed method, compared to existing implementations.

Index Terms—Storage management, performance.

1 Introduction

Flash memory has now been widely adopted in various consumer products for information storage because of its nonvolatile, shock-resistant, and power-economic nature. There are mainly two types of flash memory with different features, namely, NOR and NAND flash memory, as shown in Table 1. NOR flash memory was first introduced by Intel in 1988 and NAND flash memory was first introduced by Toshiba in 1989. While NOR flash memory is more suitable for program execution (due to the execute-in-place feature and the superior read performance), NAND flash memory is better for data storage. This work targets the design of NAND flash-memory management systems to allow vendors to trade the main-memory overhead with the system performance to best fit their needs.

A NAND flash memory chip is partitioned into blocks, where each block has a fixed number of pages and each page has a fixed size. Due to the hardware architecture, data on a flash memory chip are written in a unit of one page and no in-place update is allowed. All pages on flash memory are initially considered to be "free." When a piece of data on a page is modified, the new version of the data must be written on a free page somewhere. The pages that store old data versions are considered to be "invalid," while pages that store the most recent data versions are considered to be "valid." The status of a page, such as "valid" or "free," is stored in its spare area, while its data is stored in the data area of the page.[1] The mapping of the LBA (in a unit of one page) and its physical flash-memory address must be maintained efficiently and dynamically. Fig. 1 shows an example address translation mechanism for LBAs and their corresponding pages. The mapping problem is further complicated by the garbage collection activity in flash-memory management, in which invalid pages must be recycled. During garbage collection, the residing blocks of invalid pages must be erased before their pages become free again. As a result, data in valid pages must be copied to other free pages. Such an activity imposes further challenges on the mapping problem because of the performance and lifetime issues. The typical erasing limit of a NAND block is about 1,000,000 and the typical erasing limit of a NOR block is about 100,000. A worn-out block is marked as a "bad block" because of reliability problems.

Flash-memory management has been a critical issue in flash-memory storage system designs. Researchers and vendors have proposed various excellent designs to overcome the performance challenges under stringent resource constraints, e.g., the implementation designs and specifications from the industry, such as [2], [3], [9], [11], [16]. In particular, Wu and Zwaenepoel [17] proposed a copy-on-write scheme and a concept of page remapping to provide a transparent in-place update. Chang and Kuo [5], [6] proposed effective LBA mapping methods for multibank and large-scale flash memory. LBA mapping was also explored in a large unit, i.e., one block, to save the main-memory overhead for flash-memory management [14]. While many researchers and vendors provided transparent services for user applications and file systems to access flash memory as a block-oriented device, such as [5], [6], [14], [13], many other researchers considered the implementations of native flash-memory file systems, such as a journaling or journaling-like file system [1], [10]. In garbage

- *J.-W. Hsieh is with the Department of Computer Science and Engineering, National Taiwan University of Science and Technology, No. 43, Keelung Rd. Sec. 4, Taipei, Taiwan 106, ROC. E-mail: jenwei@mail.ntust.edu.tw.*
- *Y.-L. Tsai is with Realtek Semiconductor Corp., No. 2, Innovation Road II, Hsinchu Science Park, Hsinchu 300, Taiwan, ROC. E-mail: toronna@gmail.com.*
- *T.-W. Kuo and T.-L. Lee are with the Department of Computer Science and Engineering, National Taiwan University, No. 1, Roosevelt Rd. Sec. 4, Taipei, Taiwan 106, ROC. E-mail: {ktw, tl_lee}@csie.ntu.edu.tw.*

Manuscript received 14 Apr. 2006; revised 31 Oct. 2006; accepted 29 Jan. 2007; published online 4 Apr. 2008.
Recommended for acceptance by J. Becker.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0144-0406.
Digital Object Identifier no. 10.1109/TC.2008.61.

1. The typical size of a spare area is 16 bytes and the typical size of a data area is 512 bytes. Recently, some kinds of flash memory, e.g., Samsung K94G08Q0M, do have 2 Kbyte data areas and 64 byte spare areas for storage systems of a large capacity.

0018-9340/08/$25.00 © 2008 IEEE Published by the IEEE Computer Society

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

TABLE 1
Comparisons of NOR and NAND Flash Memory

	NOR	NAND
Cost (per MB)	$2	$0.5
Density	low	high
Access Type	linear access (close to RAM access)	sector read/write (almost like a disk)
Interface	memory interface	I/O only
eXecute In Place (XIP)	yes	no
Read/Write Performance (per page)	fast read (14.4us) slow write (3.52ms)	slow read (35.9us) fast write (226us)
Erase Time (per block)	1.2s	2ms
Erase Cycles	10,000 - 100,000	100,000 - 1,000,000
Life Span	less than 10% of the life span of NAND	over 10 times more than NOR

collection, various excellent works have been done in past years, e.g., [4], [5], [6], [12], [13], [17].

This paper proposes a configurable mapping method that could trade the main-memory overhead with the system performance under the best needs of vendors. Different from other static configuration implementations, e.g., FTL and NFTL, the objective of this work is to provide vendors flexibility to configure their address translation implementation based on the stringent main-memory space constraint in the storage of mapping tables. The address translation mechanism can be tuned up from a fine-grained one, such as a page-level mapping, to a coarse-grained one, such as a multiblock-level mapping. The finer the address translation mechanism is, the larger the table storage is. With the configurable implementation, vendors can simply set parameters of the implementation to meet the main-memory space and performance constraints, instead of code modifications. A series of experiments is conducted to provide insights on different configurations and the proposed method, compared to existing implementations.

The rest of this paper is organized as follows: In Section 2, the motivation of this paper is presented. Section 3 presents CNFTL and its overhead analysis. Section 4 addresses implementation issues. Section 5 demonstrates the capability of CNFTL compared with existing implementations. Section 6 is the conclusion.

2 Motivation

The mapping methods for flash-memory management could be roughly partitioned into two types: *page-level address mapping* and *block-level address mapping* [14]. The main design issue is the trade-off between the runtime performance and the main-memory space requirements for housekeeping information. Page-level address mapping provides a one-to-one and onto mapping between each LBA and a physical address, such as a tuple (block number, page number) [5]. An address translation table is usually stored in the main memory for efficient translation of any given LBA [5], [15], [17]. Because an LBA could be mapped to virtually any physical address, each LBA has an entry in the table such that a moderate-size flash-memory storage system (e.g., 512 Mbytes) would have a huge size (e.g., 3 Mbytes) address translation table.

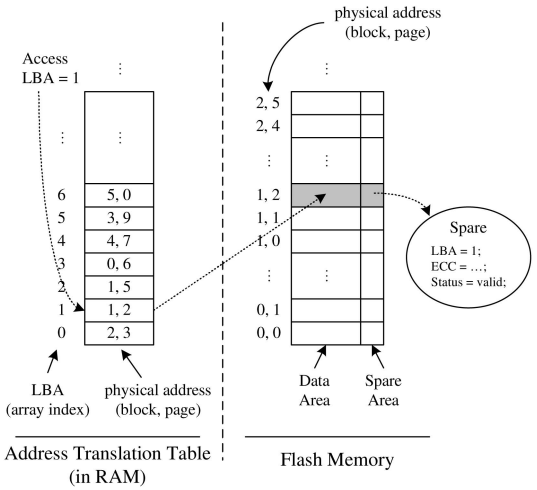

Fig. 1. An address translation mechanism for LBAs and pages.

Block-level address mapping is proposed to resolve the main-memory space requirements problem in page-level address mapping. Since each block has a fixed number of pages, an intuitive design of block-level address mapping, e.g., [14], is done by mapping an LBA to the page with the corresponding offset in the block (e.g., $LBA \bmod \pi$, where π is the number of pages per block). Such a design could also be extended by having flexibility in writing the data of an LBA to any page in the mapped block, e.g., [14], or having a one-to-many mapping between an LBA and multiple pages, e.g., [3]. A well-known one-to-many mapping implementation NFTL was proposed by Ban and Hasharon [3], in which each LBA is mapped to a logical block and a logical block has a chain of physical blocks, i.e., the real blocks in flash memory. An LBA is mapped to the page with the corresponding offset in each of the mapped physical blocks. A write of an LBA would involve a linear search of a free page in the mapped physical blocks and a read would have the same overhead. Since the required size of the address translation table is significantly reduced (e.g., 128 Kbytes for 512-Mbyte flash memory), similar block-level address mapping approaches are popular in the industry.

This research is motivated by the needs of a configurable mapping method for the mapping of LBAs and their physical addresses. It is not simply because of the needs in trading the performance and the main-memory space requirements for housekeeping information. It is also because different vendors would have different hardware requirements in implementations, such as the selection of controllers, main-memory sizes, and even flash-memory sizes and product sectors. This paper proposes a new kind of flash translation layer, called *Configurable NAND Flash Translation Layer* (CNFTL), which provides a way for different vendors to tune up the system for their needs. Furthermore, CNFTL can also be considered for many flash-memory storage systems that prefer sequential writes, such as those based on *multilevel cell (MLC)* flash memory.

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

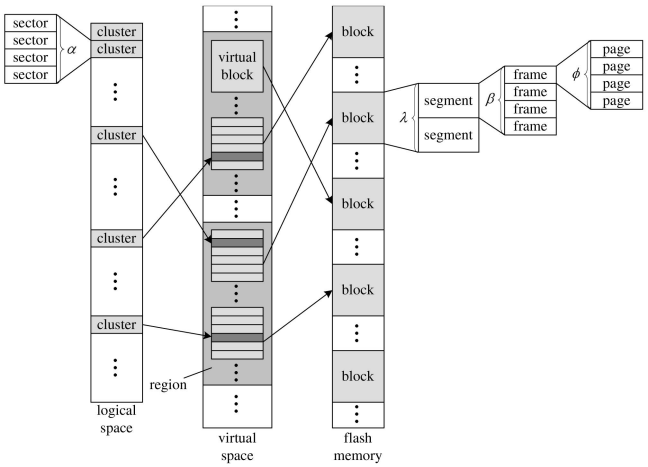

Fig. 2. The relationship among regions, segments, frames, and clusters.

3 Configurable NAND Flash Translation Layer

3.1 Overview

In CNFTL, four management units, i.e., cluster, region, segment, and frame, are introduced to provide configurable implementations. *Cluster* is used to manage the logical space. *Region* is designed for mapping from the logical space to the physical space. *Segment* and *frame* are introduced to manage the physical space. Each cluster has α sectors. It serves as a preferred access size of each read/write operation, where the access size under NFTL or FTL is fixed, i.e., a page. (It also helps in the reduction of table storage.) The LBA space is divided into regions, where a region consists of γ virtual blocks ($\gamma \geq 1$). A cluster is mapped to a region and a virtual block of the region by some formula. Each virtual block can be mapped to any physical block of flash memory such that a cluster can be virtually mapped to any frame of flash memory, where flash memory is divided into frames. The larger a region is, the larger the number of virtual blocks to serve as candidates in garbage collection will be. Under NFTL, each virtual block can be mapped to a primary (physical) block and a replacement (physical) block. The set of candidate blocks for garbage collection is that of the entire flash memory. In summary, a region is a unit to restrict the candidate space/overhead in garbage collection, compared to NFTL or FTL. The region concept (and its corresponding Block Table (BT)) also provides one-to-one mapping between virtual blocks and physical blocks to improve the flash-memory space utilization, compared to NFTL.

In the design, each physical block is divided into λ segments ($\lambda \geq 1$) and each segment is of β contiguous frames ($\beta \geq 1$). The mapping of a cluster to a segment of a physical block is done by the Cluster Table (CT). However, which frame of the corresponding segment is mapped by a cluster is done by a linear search. The mapping of an LBA and a valid page of the corresponding replacement block under NFTL is also done by a linear search to save NFTL table storage. Compared to NFTL, the β value limits the linear search time (of a cluster in a segment) in address translation under CNFTL. Each *frame* is composed of ϕ pages, where $\phi \geq 1$. The size of a *frame* is equal to that of a *cluster* ($\phi = \alpha/\sigma$, where σ is the number of sectors per page). The sizes of these four units are set at the time while flash memory is formatted and cannot be changed until the next formatting time. Fig. 2 illustrates the relationship among cluster, region, segment, and frame, where the page's size is equal to the sector's and $\alpha = 4$, $\beta = 4$, $\lambda = 2$, and $\phi = 4$, respectively.

TABLE 2
Symbol Definitions

	Symbol	Description
Hardware Parameters	π	the number of pages per block
	σ	the number of sectors per page
	δ	the total number of physical blocks in flash memory
System Parameters	α	the number of sectors per cluster
	β	the number of frames per segment
	γ	the number of blocks per region
	ρ	the number of spare free physical blocks
Dependent Parameters	ω	the total number of virtual blocks
	ϕ	the number of pages per frame ($= \alpha/\sigma$)
	φ	the number of pages per segment ($= \beta \times \phi$)
	λ	the number of segments per block ($= \lfloor \pi/\varphi \rfloor$)
	μ	the total number of regions in flash memory ($= \lfloor \omega/\gamma \rfloor$)

To have CNFTL function well, dedicated tables must be introduced. As mentioned above, the *CT* keeps the mapping from clusters to segments. The *BT* maintains the mapping from virtual to physical blocks. Both mappings are one-to-one and onto and could be changed over time. With the *Free Segment Table* (FST), the searching time for a free frame in a region could be greatly reduced. The *Block Status Table* (BST) is used to record the status of each physical block in flash memory, such as "free," "used," and "reserved." With the *BST*, CNFTL can find free physical blocks rapidly and avoid using bad physical blocks.

Cluster is the basic read/write operation unit in CNFTL. A read operation proceeds as follows: First, CNFTL finds the segment where the requested cluster resides by looking up the *CT* and the *BT*. Then, it searches throughout the segment for the frame that contains the valid data of the requested cluster. Finally, CNFTL could read the required data from the frame. A write operation proceeds as follows: First, CNFTL obsoletes the frame which contains the old version, if any, of the requested cluster. Then, it allocates a free frame from the segment by looking up the *FST*. A sequential search for a free frame might be initiated in the same region if the segment has no free frame. Finally, CNFTL could write data to the free frame. These operations are discussed in detail in Section 3.3.

Before further discussing the proposed mechanism, some symbols should be defined first. These symbols are introduced for clarity and are listed in Table 2. The first three symbols are hardware parameters of flash memory, which are constants. The next four symbols are system parameters and could be configured at the time when flash memory is formatted. Note that ρ is not for the performance consideration in address translation. Instead, it is used to determine the proper number of spare free physical blocks for reliability considerations. In summary, only three parameters, α, β, and γ, are used to have trade-off between performance and table storage under CNFTL. α is to determine a proper access size of a read/write operation. Compared to NFTL, β limits the linear search time (of an

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

LBA/cluster in a segment) in address translation, and γ manages the candidate space/overhead in garbage collection. The values of the last five symbols could be derived from the values of the above-defined symbols.

3.2 A CNFTL Framework

3.2.1 Cluster, Region, Segment, and Frame

In CNFTL, the major management unit in the logical space is the cluster. Suppose a cluster is made up of α sectors, the total number of clusters, represented as θ, in the logical space could be derived from $\omega \times \lfloor(\pi \times \sigma)/\alpha\rfloor$, where ω, π, σ, and α are the total number of virtual blocks, the number of pages per block, the number of sectors per page, and the number of sectors per cluster, respectively. Each cluster in the logical space is identified by "*cluster number*," ranging from 0 to $\theta - 1$. The corresponding cluster number for a sector can be determined from its LBA by

$$ClusterNumber = \lfloor LBA/\alpha \rfloor. \quad (1)$$

In the virtual space, flash memory is conceptually divided into μ regions, each of which is identified by "*region number*," ranging from 0 to $\mu - 1$. Each region consists of γ virtual blocks, each of which is identified by "*virtual block number*," ranging from 0 to $\gamma - 1$. CNFTL dynamically assigns, on demand, a physical block to a virtual block. Each physical block in flash memory is indexed by "*physical block number*," ranging from 0 to $\delta - 1$. After being assigned, the physical block cannot be assigned to any other virtual block until it is erased.[2]

Since physical blocks might be worn out after a fixed number of erasures, for fault tolerance concerns, the total number of physical blocks that CNFTL can actually use might be less than δ. A fixed number of spare free physical blocks, denoted as ρ, is reserved to store data copied from bad blocks. ρ should be set such that $(\delta - \rho) \bmod \gamma = 0$ so as to have blocks evenly partitioned among regions. In other words, the initial number of spare free physical blocks the system maintained should be equal to ρ. In CNFTL, ρ can only be configured at the time when flash memory is formatted. The total number of virtual blocks ω is set as $\delta - \rho$.

Each physical block is further divided into λ segments. Since a region contains γ virtual blocks, there are $\gamma \times \lambda$ segments in a region. Each segment in a region is identified from "*segment number*," ranging from 0 to $\gamma \times \lambda - 1$. The virtual block in the region a segment belongs to can be identified from its "*segment number*":

$$VirtualBlockNumber = \lfloor SegmentNumber/\lambda \rfloor. \quad (2)$$

Each segment consists of β frames and each frame consists of ϕ pages. The sizes of a frame and a cluster must be the same. It means a segment can store β clusters.

3.2.2 Cluster Table, Block Table, Free Segment Table, and Block Status Table

The CT maintains in which segment each cluster resides and is indexed by cluster number. Although a specific segment must be identified by the combination of the region number and the segment number, keeping the segment number is enough for CNFTL. This is because the corresponding region number can be calculated from its cluster number by

$$RegionNumber = ClusterNumber \bmod \mu. \quad (3)$$

The virtual block number of a segment number is derived by (2). The mapping from virtual blocks to physical blocks is maintained in the BT, which is a two-dimensional table. The first dimension is indexed by the region number and the second dimension is indexed by the virtual block number. Each entry in BT records the physical block number of the physical block which is assigned to the corresponding virtual block. For each region, the FST records the segment number of a segment which contains at least one free frame and FST is indexed by the region number. Every physical block in flash memory has its own status ("free," "used," and "reserved"), which is recorded in the BST. Status "free" means that the physical block is available to be assigned to a virtual block of any region. Status "used" indicates that the physical block has already been assigned to a virtual block. Status "reserved" suggests that the physical block is either a bad physical block or used for other purposes.

When CNFTL is required to find a sector through an LBA, it first translates the given LBA to the cluster number by (1). With this cluster number, the corresponding segment number and region number can be obtained. The segment number is looked up in CT and the region number is calculated from (3). CNFTL then determines, by (2), in which virtual block in the region the segment resides. After getting the region number and the virtual block number, CNFTL searches for the corresponding physical block number in BT. Finally, CNFTL searches the segment sequentially for the requested sector.

Fig. 3 illustrates how CNFTL finds the corresponding segment through an LBA. Suppose $\alpha = 4$, $\mu = 512$, $\lambda = 2$, and CNFTL is required to find the cluster which contains the sector with *LBA 2,093*. The corresponding cluster number is 523 since $\lfloor 2{,}093/4 \rfloor = 523$. Then, CT is referenced to find in which segment *Cluster 523* resides. As shown in the figure, $CT[523] = 7$; meanwhile, the region number is 11 ($523 \bmod 512 = 11$) and the virtual block number is 3 ($\lfloor 7/2 \rfloor$). By looking up $BT[11][3]$, the location where the requested sector resides in (i.e., *Physical Block 73*) is obtained. Since $7 \bmod 2 = 1$, the sector can be found in *Segment 1* of *Physical Block 73* in flash memory.

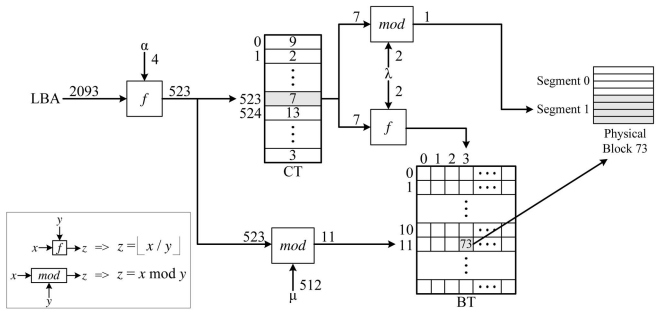

Fig. 3. An example of mapping from an LBA to a segment.

2. All issues about erasing a physical block during *garbage collection* are discussed in detail in Section 3.3.4.

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

3.2.3 Main-Memory Space Requirements

With the notions of cluster, segment, and region, the required main-memory usage of CNFTL is quite small compared with that of page-level address mapping. The main-memory space requirements of CNFTL are dominated by the tables mentioned in Section 3.2.2, namely, CT, BT, FST, and BST. This section discusses the size of these tables. To facilitate the following discussion, κ is defined as the number of segments per region, which is equal to $\gamma \times \lambda$.

For CT, $(\lfloor \log \kappa \rfloor + 1)$ bits are needed for each cluster to record in which segment it resides. Note that a nonexisting cluster in CNFTL is marked by setting all of the bits of its corresponding entry to "1." Hence, the size of CT can be calculated by

$$SizeOfCT = ((\theta \times (\lfloor \log \kappa \rfloor + 1))/8) \text{ bytes},$$

where θ, as defined in Section 3.2.1, denotes the total number of clusters in logical space. The last division is used to transfer the unit from bit to byte. This equation indicates that the size of CT is impacted by the values to which α, β, γ, and ρ are set.

For BT, $(\lfloor \log \delta \rfloor + 1)$ bits are needed for each virtual block of each region to record its corresponding physical block number. Note that any virtual block of a region that has not been assigned a physical block is marked by setting all of the bits of this virtual block's corresponding entry to "1." Hence, the size of BT can be calculated by

$$SizeOfBT = ((\omega \times (\lfloor \log \delta \rfloor + 1))/8) \text{ bytes}.$$

This equation indicates that the size of BT is impacted by the value to which ρ is set.

For FST, $\lceil \log \kappa \rceil$ bits are needed for each region to record which segment has any free frame. Hence, the size of FST can be calculated by

$$SizeOfFST = ((\mu \times \lceil \log \kappa \rceil)/8) \text{ bytes}.$$

This equation indicates that the size of FST is impacted by the values to which α, β, γ, and ρ are set.

For BST, only 2 bits are needed for each physical block to record its status ("free," "used," and "reserved"). Hence, the size of BST can be calculated by

$$SizeOfBST = ((2 \times \delta)/8) \text{ bytes}.$$

The main-memory space requirements of the tables could be shown by the following example: Let $\alpha = 4$, $\beta = 4$, $\gamma = 16$, $\rho = 16$ and the size of flash memory be 512 Mbytes. Based on equations shown in the previous sections, $\omega = 32{,}752$, $\theta = 262{,}016$, $\mu = 2{,}047$, and $\kappa = 32$. The main-memory space requirements of the tables are $SizeOfCT = 196{,}512$ bytes, $SizeOfBT = 65{,}504$ bytes, $SizeOfFST = 1{,}280$ bytes, and $SizeOfBST = 8{,}192$ bytes. The total main-memory space requirement under the CNFTL is about 271 Kbytes, compared with 128 Kbytes for block-level address mapping and 3 Mbytes for page-level address mapping.

3.3 Operations

3.3.1 Processing of One Request

This section describes basic operations in CNFTL, including $ReadOneCluster$, $WriteOneCluster$, and $GarbageCollection$. Procedure 1 shows how CNFTL processes a request. When a request arrives, CNFTL first determines $StartCN$ and $EndCN$ (Steps 1-2), where $StartCN$ and $EndCN$ are the cluster numbers of the first and the last requested clusters. CNFTL then checks the type of the request. If the request type is "READ," CNFTL reads one cluster at a time until all needed clusters are read from flash memory (Steps 4-6). During each iteration, $ReadOneCluster$ is invoked to read *Cluster CN* (Step 5). Otherwise, the request type must be "WRITE" and CNFTL writes a cluster at a time until all clusters in question are written to flash memory (Steps 8-10). Similarly, $WriteOneCluster$ is invoked to write *Cluster CN* (Step 9) during each iteration. $GarbageCollection$ is invoked internally when extra free space is required and is discussed in Section 3.3.4.

Procedure 1 $ProcessingOneRequest(Req)$

```
1: StartCN ⇐ Req.StartLBA/α
2: EndCN ⇐ (Req.StartLBA + Req.Length − 1)/α
3: if Req.Type = READ then
4:     for CN = StartCN to EndCN do
5:         ReadOneCluster(CN)
6:     end for
7: else {Req.Type = WRITE}
8:     for CN = StartCN to EndCN do
9:         WriteOneCluster(CN)
10:    end for
11: end if
```

3.3.2 Reading of One Cluster

The procedure *ReadOneCluster* is illustrated in Procedure 2. Input CN is the cluster number indicating the cluster which CNFTL wants to read. CNFTL first looks up CT to get the corresponding segment number SN (Step 1). If SN is equal to -1, it means that *Cluster CN* has not yet been written to flash memory and CNFTL returns all 0s (Step 3). Otherwise, CNFTL tries to find the valid data of *Cluster CN* and reads them from flash memory (Steps 5-18). In finding the valid data of *Cluster CN*, CNFTL first determines the physical address area where *Cluster CN* resides (Steps 5-8). Here, VBN and PBN are the virtual and physical block numbers of the cluster.

Procedure 2 $ReadOneCluster(CN)$

```
1: SN ⇐ CT[CN]
2: if SN = −1 then
3:     return all 0s {The read request is failed}
4: else
5:     RN ⇐ CN mod μ
6:     VBN ⇐ ⌊SN/λ⌋
7:     PBN ⇐ BT[RN][VBN]
8:     PN ⇐ (SN mod λ) × φ
9:     while PN < π do
10:        Spare ⇐ ReadSpareArea(PBN, PN)
11:        if Spare.Status = "valid" ∧ Spare.CN = CN then
12:            ReadPages(PBN, PN, φ)
13:            return the read data
14:        else
15:            PN ⇐ PN + φ
16:        end if
```

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

17: **end while**
18: return all 0s {The read request is failed}
19: **end if**

CNFTL then searches in *Segment SN* of *Region RN* for the frame containing the valid data of *Cluster CN* and reads them from flash memory (Steps 9-17). The spare area of the first page in a frame is read to check if the frame is "valid" and assigned to *Cluster CN* (Steps 10-11). Once the frame has been found, the data are accessed and returned (Steps 12-13). Otherwise, CNFTL moves to the next frame (Step 15). If CNFTL still cannot find the frame which contains the valid data of *Cluster CN* after reaching the segment's boundary, the read request fails and CNFTL returns all 0s to the requester (Step 18).

3.3.3 Writing of One Cluster

The procedure *WriteOneCluster* is shown in Procedure 3. Input CN is the cluster number indicating the cluster which CNFTL wants to write. When CNFTL is requested to write *Cluster CN*, it first determines the corresponding region number RN from CN (Step 1). Then, CNFTL tries to find and obsolete the old version, if any, of *Cluster CN* (Steps 2-6). The way to find the old version of *Cluster CN* is similar to Procedure 2, except that CNFTL does not read the data in the frame (Step 4). The old version of *Cluster CN* is obsoleted by marking the frame as "invalid" in the spare area of the first page in the frame (Step 5).

Procedure 3 $WriteOneCluster(CN)$
1: $RN \Leftarrow CN \bmod \mu$
2: $OldSN \Leftarrow CT[CN]$
3: **if** $OldSN \neq -1$ **then**
4: find the valid data for *Cluster CN*
5: obsolete the valid data
6: **end if**
7: $Phase \Leftarrow 1$
8: $SN \Leftarrow FST[RN]$
9: $VBN \Leftarrow \lfloor SN/\lambda \rfloor$
10: $PBN \Leftarrow BT[RN][VBN]$
11: **if** $PBN = -1$ **then**
12: find a free physical block *FreePBN*
13: $BT[RN][VBN] \Leftarrow FreePBN$
14: $PBN \Leftarrow BT[RN][VBN]$
15: **end if**
16: $PN \Leftarrow (SN \bmod \lambda) \times \varphi$
17: **while** $PN < \pi$ **do**
18: $Spare = ReadSpareArea(PBN, PN)$
19: **if** $Spare.status = ``free''$ **then**
20: $WritePages(PBN, PN, \phi)$
21: $WriteSpareArea(PBN, PN, CN)$
22: $CT[CN] \Leftarrow SN$
23: return $SUCCESS$
24: **else**
25: $PN \Leftarrow PN + \phi$
26: **end if**
27: **end while**
28: **if** $Phase = 1$ **then**
29: $FindFreeSegment(RN)$
30: $Phase \Leftarrow Phase + 1$
31: goto "Step 8"
32: **else**
33: return $FAIL$
34: **end if**

Now, CNFTL needs to find a free frame for writing the new version of *Cluster CN* (Steps 7-34). *Phase* is the flag that helps CNFTL to determine whether any free frame exists, which is described later. At first, CNFTL gets the segment number, i.e., SN, of the segment which might have any free frame (Steps 8-15). If *Virtual Block VBN* of *Region RN* has not yet been assigned a physical block, CNFTL tries to find a free physical block and assigns it to *Virtual Block VBN* of *Region RN* (Steps 11-15). Then, CNFTL searches for a free frame in *Segment SN* of *Region RN* and writes the new version of *Cluster CN* to the free frame (Steps 16-27). To find a free frame, CNFTL reads the spare area of the first page in a frame (Step 18) and checks if it is "free" (Step 19). Once a free frame is found, CNFTL writes the new version of *Cluster CN* to the frame, marks it as the "valid" frame of *Cluster CN*, and updates the corresponding entry in CT to SN (Steps 20-22). If CNFTL still cannot find a free frame after reaching the segment's boundary, procedure *FindFreeSegment*, which is discussed later, must be invoked to get free space and then CNFTL tries to write *Cluster CN* again (Steps 29-31). If CNFTL still cannot find a free frame after reaching the segment's boundary in the second run, the write request fails (Step 33).

The procedure *FindFreeSegment* is listed in Procedure 4. Input RN is the region number indicating the region where CNFTL wants to find a segment with free frames. CNFTL first checks if any virtual block in *Region RN* is not assigned a physical block (Steps 1-6). Once CNFTL finds one, it sets the corresponding entry in FST to the segment number of the first segment in the unassigned virtual block (Step 3) and then returns (Step 4). If every virtual block in *Region RN* has been assigned a physical block, CNFTL begins *garbage collection* which is discussed in Section 3.3.4 (Step 7). After garbage collection, CNFTL can get *Virtual Block VBN* of *Region RN* which has at least one free frame in *Region RN*. Then, CNFTL finds which segment in *Virtual Block VBN* contains a free frame and sets the corresponding entry in FST to the segment number of the found segment (Steps 9-15).

Procedure 4 $FindFreeSegment(RN)$
1: **for** $VBN = 0$ to $\gamma - 1$ **do**
2: **if** $BT[RN][VBN] = -1$ **then**
3: $FST[RN] \Leftarrow \lambda \times VBN$
4: return
5: **end if**
6: **end for**
7: $VBN \Leftarrow GarbageCollection(RN)$
8: $SN \Leftarrow \lambda \times VBN$
9: **for** $i = 1$ to λ **do**
10: **if** *Segment SN* of *Region RN* has at least one free frame **then**
11: $FST[RN] \Leftarrow SN$
12: return
13: **end if**
14: $SN \Leftarrow SN + 1$
15: **end for**

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

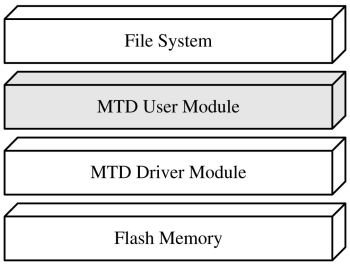

Fig. 4. An MTD subsystem architecture.

3.3.4 Garbage Collection

Garbage collection begins when a write request could not find a segment with any free frame in the corresponding region. Such a condition is detected by both failures in the searching of a free frame in the segment recorded by the FST (Steps 17-27 in Procedure 3) and in the seeking of any virtual block without physical block assignment (Steps 1-6 in Procedure 4).

The procedure *GarbageCollection* is shown in Procedure 5. Input RN is the region number indicating the region which needs garbage collection. First, CNFTL has to select a block to begin garbage collection (Step 1). There are a number of selection policies that can be used in garbage collection for CNFTL, such as [7], [8], [13]. Different selection policies result in different performances. However, the comparison among these policies is out of the scope of this paper.

Procedure 5 $GarbageCollection(RN)$

1: select *Virtual Block VBN* of *Region RN* to do garbage collection
2: $ErasePBN \Leftarrow BT[RN][VBN]$
3: find a free physical block *FreePBN*
4: $DPN = 0$
5: **for** $SPN = 0$ to $\pi - 1$ **do**
6: $Spare = ReadSpareArea(PBN, SPN)$
7: **if** $Spare.Status =$ "$valid$" **then**
8: $ReadPages(ErasePBN, SPN, \phi)$
9: $WritePages(FreePBN, DPN, \phi)$
10: $CT[Spare.CN] \Leftarrow VBN \times \lambda + \lfloor DPN/\varphi \rfloor$
11: $DPN \Leftarrow DPN + 1$
12: **end if**
13: **end for**
14: $EraseBlock(ErasePBN)$
15: $BST[ErasePBN] \Leftarrow$ "$free$"
16: $BST[FreePBN] \Leftarrow$ "$used$"
17: $BT[RN][VBN] \Leftarrow FreePBN$
18: return VBN

Suppose *Virtual Block VBN* of *Region RN* is selected for garbage collection; CNFTL could determine its corresponding physical block number $ErasePBN$ by looking up BT (Step 2). Since the data in the block are lost after erasure, CNFTL must find a free physical block and, before erasure, sequentially copy valid data from *Physical Block ErasePBN* to the free physical block (Steps 3-13). While copying valid data, the corresponding entries in *CT* also need to be updated (Step 10). After copying, CNFTL erases *Physical Block ErasePBN* and updates corresponding entries of BST and BT (Steps 14-17). Finally, CNFTL returns VBN (Step 18).

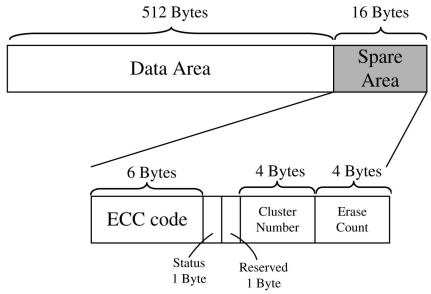

Fig. 5. The data layout of a page.

4 Implementation Remarks

CNFTL was implemented in the Memory Technology Devices (MTD) subsystem[3] of Linux.[4] The MTD subsystem is divided into the MTD driver module and the MTD user module, as shown in Fig. 4. The MTD driver module provides raw read/write/erase operations to physical memory devices for the MTD user module and the MTD user module is a high-level interface for the file system. CNFTL shall be implemented as an MTD user module due to its design goals.

The adopted implementation platform is an SMDK2410 (Samsung MCU Development Kit) development board whose microprocessor is S3C2410X. S3C2410X is a 16/32-bit RISC microprocessor using ARM920T core. SMDK2410 is comprised of a 1 Mbyte onboard NOR flash memory and a SmartMedia card[5] (SM card) slot. SM card is a kind of NAND flash memory without any controller, i.e., CNFTL can control SM card directly. Due to the space limitation of the onboard NOR flash memory, SMDK2410 boots from the SM card in the implementation. The SM card adopted is 64 Mbytes in capacity and is partitioned into 4 Mbytes (for boot images and referred to as the boot partition) and 60 Mbytes (for data storage and referred to as the data partition). Only the data partition is controlled by CNFTL in the implementation.

Fig. 5 illustrates the data layout of a page. Spare area is used for page information and data area is used for data. In the implementation, there are an *ECC code* field, a *status* field, a *cluster number* field, and an *erase count* field in spare area. The *ECC code* field records the ECC code of the data in the data area, where ECC code is used to detect/correct errors of the data from which the ECC code is derived. The *status* field records the status (free/valid/invalid) of the frame in which this page resides. The eighth byte of spare area is reserved. The *cluster number* field records the cluster number of the cluster to which the data in the data area belongs. The *erase count* field records the erase count of the physical block in which this page resides. The ECC code is recorded in the spare area of every page. The status and the cluster number of each frame are recorded in the spare area of the frame's first page because a cluster is the basic read/write unit. The erase count of each physical block is recorded in the spare area of the physical block's first page.

3. http://www.linux-mtd.infradead.org/index.html.
4. http://www.linux.org/.
5. http://www.ssfdc.or.jp/english/common/f_spec.htm.

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

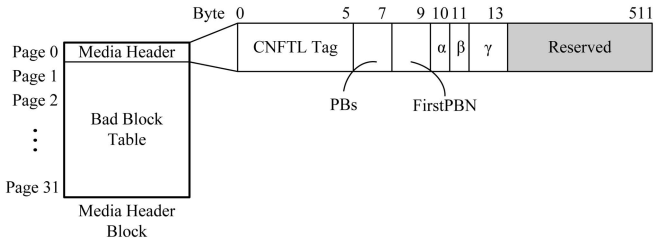

Fig. 6. The data layout of the media header block.

Similarly to the implementation of NFTL in the MTD subsystem, the implementation of CNFTL also has the *media header block* and the *spare media header block* for system recognition and bad-block management. The *media header block* and the *spare media header block* have the same contents (for fault tolerance concern) and are filled in at formatting time. Since the content of a defective bit in flash memory might turn from "1" to "0," the data in the media header block should be OR-ed with the data in the spare header block to reduce the probability of false information. Note that the format of the media header block in CNFTL is somewhat different from that in NFTL due to their dissimilar frameworks. Fig. 6 represents the data layout of the media header block. The media header block is composed of "media header" and "bad BT." Information in the media header is required at mounting time: The *CNFTL tag* field indicates that the partition is managed by CNFTL. The *PBs* field stores the number of physical blocks contained in the partition. The *FirstPBN* field records the starting physical block number (i.e., the physical block number of the first physical block) which CNFTL can use;[6] The 11th and 12th bytes store the values of α and β. The 13th and 14th bytes together store the value of γ. In the bad BT, each physical block in the partition has 1 byte to indicate whether this block is "good" or "bad." At mounting time, CNFTL first locates the *media header block* and the *spare media header block* by searching for the blocks beginning with "CNFTL tag." If no such block is found, the partition cannot be managed by CNFTL. If the blocks are found, CNFTL gets the rest of the information from the media header and scans the "bad BT" to mark bad physical blocks as "reserved" in BST. The "reserved" physical blocks cannot be used by CNFTL.

Since block-oriented devices can only handle one request at a time, requests from the file system are first inserted into their respective *request queue*, waiting to be processed. In the implementation, because flash memory is emulated as a block-oriented device, the same mechanism is also constructed. The size of the space a request accesses is fixed for a block-oriented device and is usually smaller than the cluster's size. When a request arrives, CNFTL would read/write a cluster from/to flash memory. It is observed that read/write overhead arose in CNFTL since the requested data are just a portion of a cluster. To reduce such overhead, the *cluster buffer*, which is occupied in main memory, is introduced for CNFTL. The size of the *cluster buffer* is the same as that of a cluster.

6. Those physical blocks, such as the media header block, the spare media header block, bad blocks, and boot blocks, cannot be used by CNFTL.

Data read from flash memory are first captured in the cluster buffer. With the cluster buffer, the overhead of bulk reading could be reduced. If a write request wants to partially update an existing cluster, the data of that cluster are first copied to the cluster buffer and update is performed in the cluster buffer. Otherwise, the data are written to the cluster buffer straightforwardly. The data in the cluster buffer would not be written to flash memory until the target cluster of a write request changed. Astute readers might point out that, if the cluster's size becomes larger, unnecessary writes to pages would increase, even if the notion of the cluster buffer is introduced. A simple trick is used to relieve this problem: If the content of some page in the cluster buffer is filled with "0xff," the page will not be written to flash memory.

5 Performance Evaluation

5.1 Performance Metrics and Experiment Setup

This section evaluates the performance differences of CNFTL with different configurations and another popular approach, NFTL [3], under a trace-driven simulation, where NFTL is one of the most well-known and widely adopted implementations of flash translation layer. The performance of flash translation layer was evaluated by data-access throughput. Since the data-access throughput was dominated by required read/write/erase operations, the number of these required operations was adopted as performance metrics. Under different settings of the cluster size, the region size, and the segment size for CNFTL, the impacts on the performance were explored and performances of CNFTL and NFTL are compared.

The trace of data access for performance evaluation was collected over a hard disk of a mobile PC with a 20 Gbyte hard disk, a 384 Mbyte RAM, and an Intel Pentium III 800 MHz processor [5]. The operating system was Windows XP and the hard disk was formatted as NTFS. Traces were collected by inserting an intermediate filter driver into the kernel and the duration for trace collecting was one month. The workload of the mobile PC in accessing the hard disk corresponded to the daily use of many people, i.e., Web surfing, e-mail sending/receiving, movie playing and downloading, document typesetting, and gaming.

In the experiments, a 64 Mbyte NAND flash memory with 32 pages per block and 512 bytes per page was emulated. The number of the spare free physical blocks was set to 16 $(\rho = 16)$. In other words, the logical space of the flash memory was comprised of 130,560 sectors. Let the sizes of a page and a sector be the same. The number of blocks per region, i.e., γ, was set to an adequate value so that $(\delta - \rho) \bmod \gamma = 0$. To emulate a flash memory storage system with its logical space equal to 130,560 sectors, in the trace, only the read/write operations accessing LBAs within a range of 130,560 sectors were extracted.

Suppose the number of frames with valid data was ψ and the total number of clusters was θ; the utilization was defined as ψ/θ. Fig. 7 shows the utilization variation of the trace. The X-axis is cumulative and is the number of sectors the file system requested to write. The Y-axis is the utilization. The utilization kept increasing as long as new

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

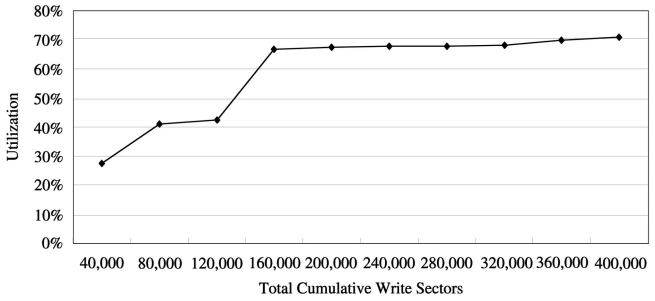

Fig. 7. The utilization of the logical space after writing various numbers of sectors.

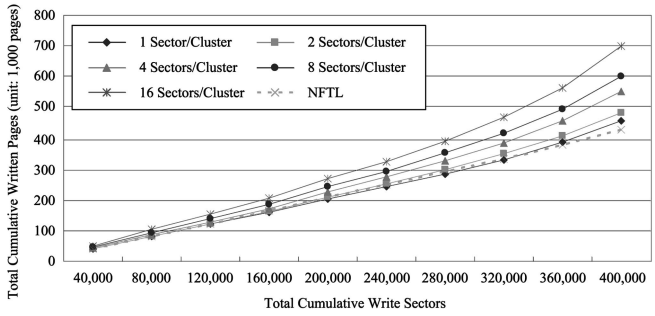

Fig. 8. The number of written pages under various cluster sizes.

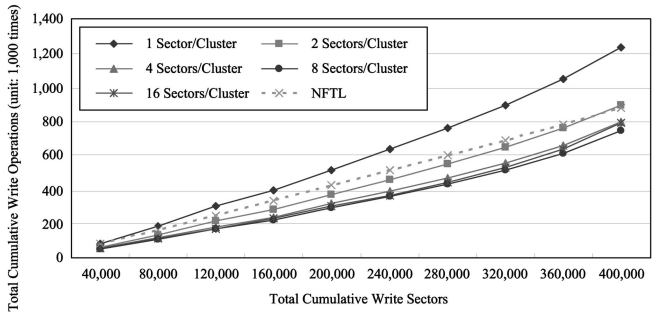

Fig. 9. The number of write operations under various cluster sizes.

data were written to flash memory. If the write request was just to update previously written data, the utilization stayed unchanged. As shown in Fig. 7, the slope of utilization variation smoothed after writing 160,000 sectors and the maximum utilization was 70.83 percent.

5.2 CNFTL, NFTL, and Page-Level Mapping

5.2.1 The Impacts of the Cluster Size

The first experiment was to compare the performance differences between *CNFTL under various cluster size settings* and *NFTL*. To avoid disturbance, the settings of the segment size β and the region size γ were fixed for CNFTL. In this experiment, $\beta = 2$ and $\gamma = 8$, while the cluster size α ranged from 1 to 16 sectors. Fig. 8 illustrates the cumulated number of written pages due to sustained write requests for CNFTL under various cluster sizes and for NFTL. For CNFTL, when the cluster size became larger, CNFTL needed to write more pages to flash memory. This was because that cluster was the basic write operation unit in CNFTL and the larger cluster induced extra page writes. For NFTL, since its basic write operation unit was page, NFTL wrote fewer pages than most configurations of CNFTL did. Both CNFTL

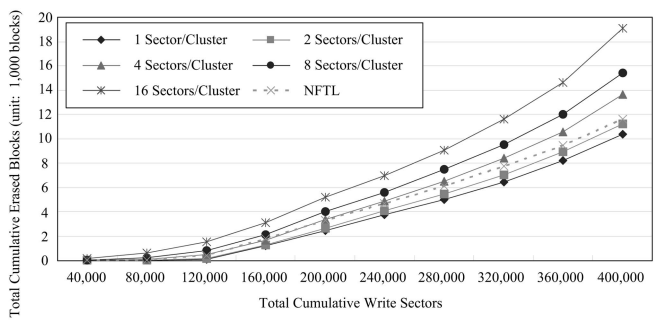

Fig. 10. The number of erased blocks under various cluster sizes.

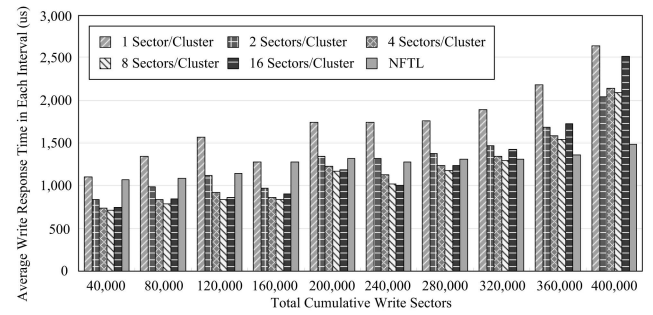

Fig. 11. The write response time under various cluster sizes.

and NFTL had to write management information to the spare area for keeping track of the written data. CNFTL wrote management information to the spare area of the first page in the frame after cluster data was written or before it was updated. For a fixed amount of data requested for write, the required number of write operations for CNFTL increased when the cluster size became smaller, as shown in Fig. 9. The results shown in Fig. 9 were quite different from those in Fig. 8 because of extra writes to spare areas.

Fig. 10 shows the number of blocks being erased due to sustained write requests for CNFTL under various cluster sizes and for NFTL. As the cluster size became larger, a region would rapidly run out of its free pages. As a result, the number of blocks erased due to sustained write requests increased with the cluster size. Although enlarging the cluster size increased the number of erase operations, it also reduced the number of write operations. Since both erase operations and write operations affected the write performance, setting the cluster size too large or too small could negatively affect the write performance, as illustrated in Fig. 11. To reduce main-memory requirement, NFTL introduced extra page reads to find the page for a write request. As a result, write response times of NFTL for sustained write requests depended not only on numbers of erase operations and write operations but also on the number of extra page reads. It was the reason why the write performance of NFTL was not as good as expected, compared to those in Figs. 8, 9, and 10. Fig. 12 shows the corresponding write throughput of CNFTL under various cluster sizes and NFTL. The main-memory space requirements required by CNFTL under various cluster sizes and NFTL are illustrated in Fig. 13. When the cluster size was enlarged, the sizes of *CT* and *FST* reduced and the main-memory space required by CNFTL shrank as well.

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

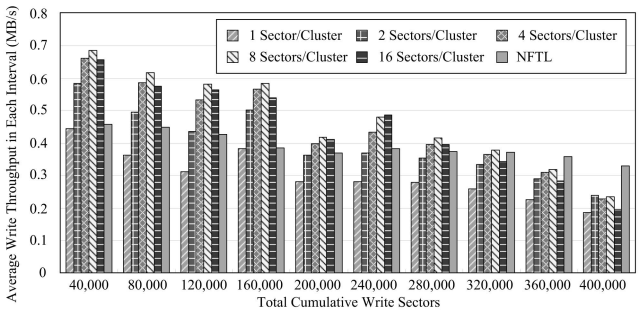

Fig. 12. The write throughput under various cluster sizes.

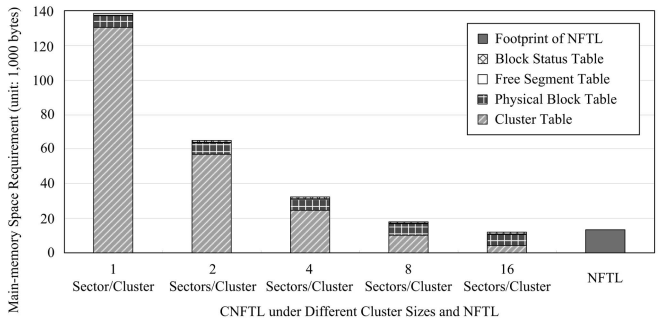

Fig. 13. The main-memory space requirements under various cluster sizes.

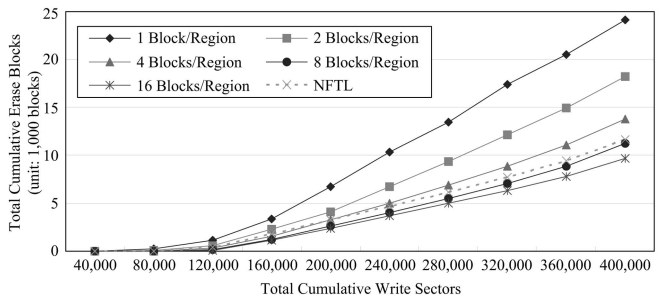

Fig. 14. The number of erased blocks under various region sizes.

5.2.2 The Impacts of the Region Size

The second experiment evaluates the impacts of different region size settings for CNFTL. Similarly, the settings of the cluster size α and the segment size β were fixed to avoid disturbance. In this experiment, $\alpha = 2$ and $\beta = 2$, while the region size γ ranged from 1 to 16 blocks. Fig. 14 shows the number of blocks being erased due to sustained write requests for CNFTL under various region sizes and for NFTL. For CNFTL, when the region size became larger, the time when a region ran out of its free space would be postponed. More valid clusters in the region would turn into invalid ones before garbage collection was invoked. Thus, the number of free clusters obtained from garbage collection in the region would increase. For this reason, the time of the next garbage collection for the region would be postponed further and such circumstances would repeat. In other words, when the region size got larger, the time interval between every two consecutive garbage collections for a region would be prolonged and the number of blocks being erased for a region would decrease. As shown in Fig. 14, the number of erased blocks decreased as the region size was enlarged.

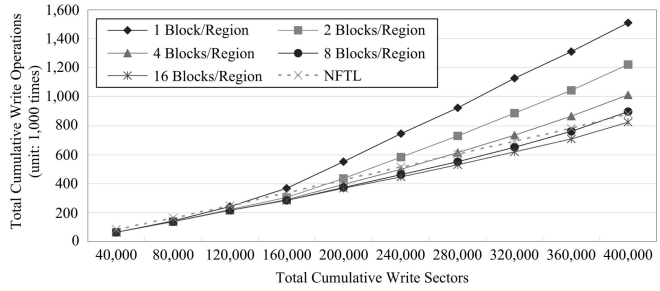

Fig. 15. The number of write operations under various region sizes.

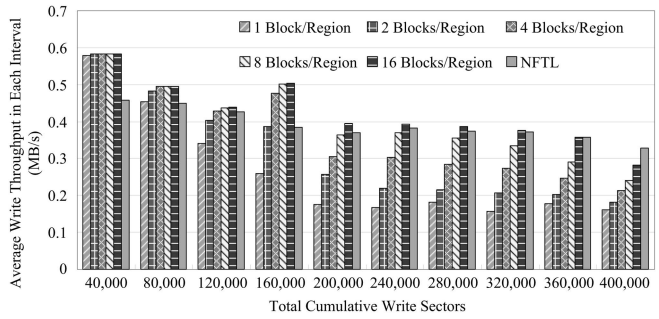

Fig. 16. The write throughput under various region sizes.

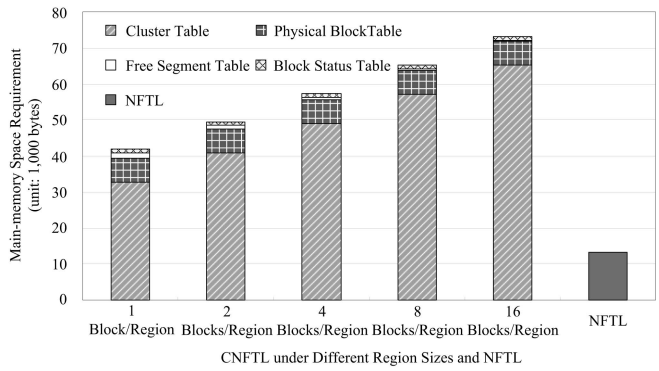

Fig. 17. The main-memory space requirements under various region sizes.

While the number of erased blocks decreased, the number of valid-data copying during garbage collection decreased as well. Therefore, the total number of write operations was reduced when the region size increased, as shown in Fig. 15. To have a concrete idea about the impact of different region sizes for CNFTL over the write performance, Fig. 16 compares the write throughput of CNFTL under various region sizes and NFTL. Fig. 17 illustrates the main-memory space requirements for CNFTL under various region sizes and for NFTL. Enlarging the region size can diminish the size of the *FST* but raise the size of the *CT*. Unfortunately, the amount of size that *FST* diminished was less than the amount of size that *CT* increased. Although enlarging the region size could achieve better performance, it raised the main-memory space requirements.

5.2.3 The Impacts of the Segment Size

The third experiment evaluates the performance differences for CNFTL under various segment size settings and for NFTL. To avoid disturbance, the settings of the cluster size α and the

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

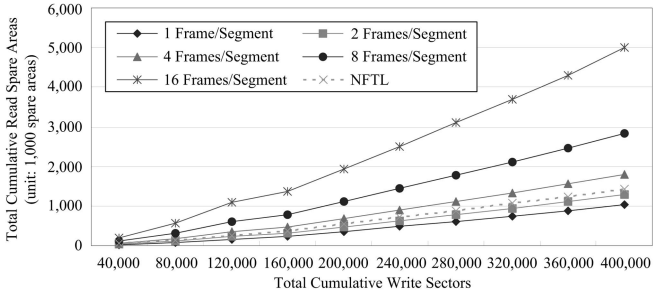

Fig. 18. The number of read operations on spare areas under various segment sizes.

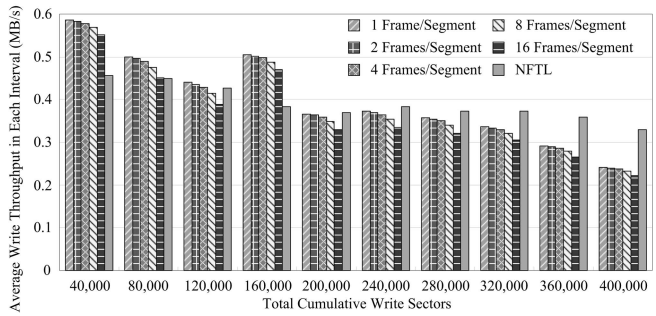

Fig. 19. The write throughput under various segment sizes.

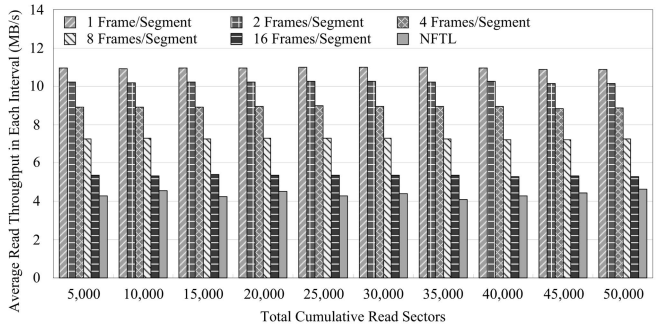

Fig. 20. The read throughput under various segment sizes.

region size γ were fixed for CNFTL. In this experiment, $\alpha = 2$ and $\gamma = 8$, while the segment size β ranged from 1 to 16 frames. Regarding the write performance, changing the segment size only affected the number of read operations on spare areas. When CNFTL needed to find a cluster, it first looked up the *CT* to determine in which segment the cluster resides and then sequentially searched every frame in the segment until the frame containing the valid data of the required cluster was found. If the segment size became larger, the average number of frames examined increased before CNFTL found the required cluster. In other words, the number of the required read operations on spare areas increased as the segment size was enlarged. Fig. 18 supports this argument.

Fig. 19 evaluates the overall write performances of CNFTL under various segment size settings and NFTL via the write throughput. Compared with the impacts of different cluster sizes or different region sizes for CNFTL on write performance, the impact of different segment sizes for CNFTL on write performance was minor. On the other hand, the impact of different segment sizes for CNFTL on read performance was more significant, as shown in Fig. 20.

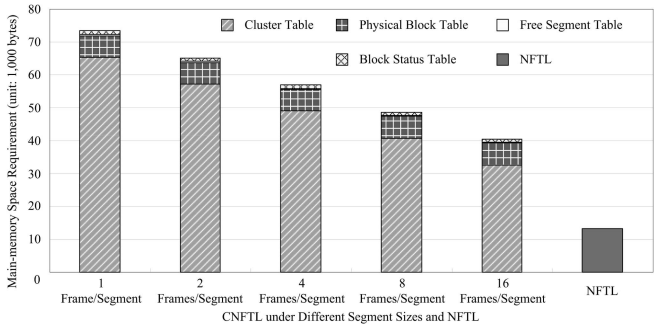

Fig. 21. The main-memory space requirements under various segment sizes.

Although enlarging the segment size deteriorates the read and write performances, it can reduce the main-memory space requirements. Both the sizes of the *CT* and the *FST* reduced as the segment size became larger. Fig. 21 illustrates the impacts of different segment size settings on the main-memory space requirements.

5.2.4 Performance Limits

The proposed method aims at providing a trade-off between the performance in address translation and the required table storage space. We introduce extra mapping tables and mapping calculation, compared to that of FTL or NFTL, to trade the performance with the table storage. The calculation overhead is fairly limited because it is done by a few mod, /, or a floor operation. The introduction of extra tables results in overhead in another way: Since each read/write operation is of one cluster, different values of α (that is, the size of a cluster) do introduce different overhead to the system. When α is large and smaller files are considered, the throughput of the proposed method will decrease, in general, because the access size of each operation is large. In addition to clusters, the introduction of segments and regions in the proposed method is mainly for the flexibility in mapping of LBAs and their physical addresses, where selected bits of LBAs are used to do address mapping. Compared to NFTL, they are extra information to speed up address translation, where NFTL depends on linear scanning of replacement blocks when LBAs are not found in primary blocks.

Two configurations of CNFTL, called CNFTL-A and CNFTL-B, were chosen to provide insights in system designs. The settings of CNFTL-A included $\alpha = 2$, $\beta = 4$, and $\gamma = 1{,}020$ and the settings of CNFTL-B included $\alpha = 8$, $\beta = 4$, and $\gamma = 1{,}020$. (Please refer to Table 2 for symbol definitions.) γ was set to 1,020 since the improvement on the write performance, due to the larger region size, would saturate when a region contained more than 1,020 blocks. The major difference between CNFTL-A and CNFTL-B was on the setting of the cluster size α, where a cluster is the basic read/write operation unit in CNFTL. As discussed in Section 5.2.1, a large value for α could reduce the number of extra writes to spare areas. However, CNFTL might need to read/write more pages for each read/write request. On the other hand, a small value for α would result in a better space utilization at the cost of the increasing in the number of write operations (due to writes of management information). In this experiment,

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

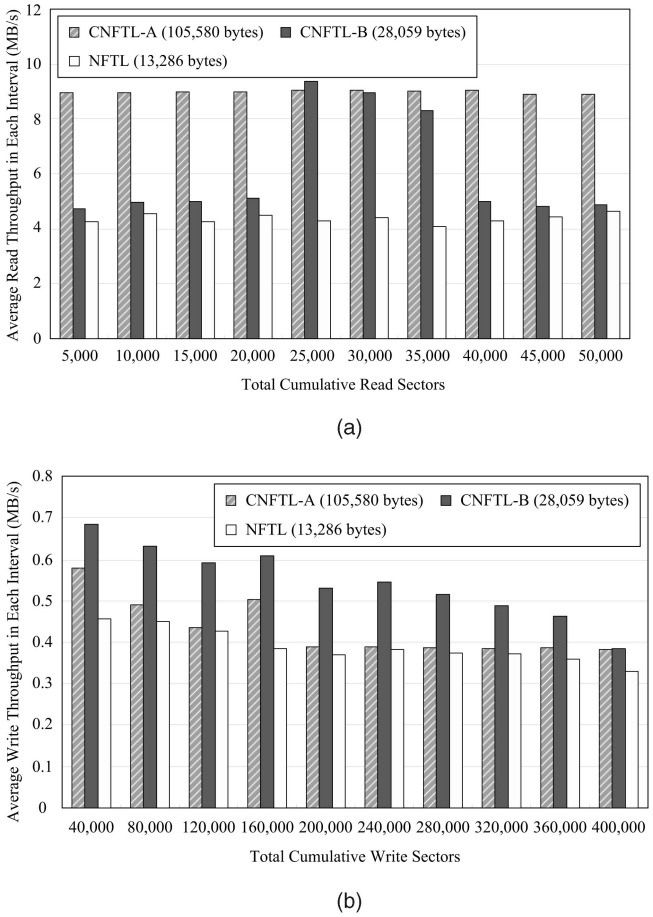

Fig. 22. Performance comparisons between CNFTL and NFTL. (a) The average read throughput. (b) The average write throughput.

β was set to 4 because it has advantages in performance and memory-space overheads.

Fig. 22 contains a performance comparison of NFTL and the proposed method with two different configurations (i.e., CNFTL-A and CNFTL-B) to have more insights. The main-memory space requirements of CNFTL-A, CNFTL-B, and NFTL were listed in the legends: The main-memory space requirements of CNFTL-A, CNFTL-B, and NFTL were 105,580, 28,059, and 13,286 bytes, respectively. The table storage of CNFTL-A was much more than that of CNFTL-B because the α value of CNFTL-A was smaller than that of CNFTL-B. The write throughput of CNFTL-B was better than that of CNFTL-A because there were a smaller number of write operations under CNFTL-B (please see Fig. 9 on the impacts of the cluster size). The throughput gap between CNFTL-A and CNFTL-B decreased when the number of cumulative write sectors was more than 200,000 because of more garbage collection activities due to the occurrences of more updates (please see statements related to Fig. 7). The read throughput of CNFTL-A was, in general, better than that of CNFTL-B because the amount of data requested by read operations in the trace was often less than the cluster size of CNFTL-B. As readers might notice, the read throughput of CNFTL-A was similar to that of CNFTL-B when the number of sectors read in the trace was between 25,000 and 35,000. We had some analyses over the trace and found out that the amount of data requested by read operations tended to be large during that interval. In general, the write throughput favored large clusters (with a large α value). The read throughput was sensitive to the average size of read operations. Although the main-memory space required by either CNFTL-A or CNFTL-B was more than that required by NFTL, their main-memory space requirements seemed affordable in many implementations.[7]

Note that CNFTL becomes FTL when each cluster is one page (i.e., one sector), one segment is one cluster, and there is only one region (that consists of all blocks). That is, $\alpha = \beta = 1$ and γ is the total number of blocks (i.e., $\rho = 0$). In this way, the table storage of CNFTL will be as huge as that of FTL, but the address translation performance is the best among any address translation mechanisms. CNFTL becomes another extreme alternative when each cluster consists of all pages, one segment is one cluster, and there is only one region (i.e., the cluster). This configuration has nearly no space need because every LBA is mapped to the same cluster and a linear search is required to go over the cluster to find the corresponding page. The address translation performance is infeasible and the worst because of the linear search. Such a configuration is worse than NFTL but needs no table storage. There are trade-offs in parameter settings based on the needs of the address translation performance and the table storage. Note that CNFTL with γ over 1,024 is close to CNFTL with one single region only in the experiments. The setting of α and β has more impact on the read/write throughput, the table storage needs, and the number of erased blocks, as addressed in Sections 5.2.1 and 5.2.3.

6 Conclusions

This paper has proposed a CNFTL for the emulation of block-oriented devices. It provides vendors with better flexibility in trading the main-memory overhead with the system performance to fit their best needs. Flexibility in mapping any LBA to virtually any page in any block on flash memory is realized. In order to achieve that goal and to simultaneously manage the main-memory space requirements, several mapping tables and LBA calculation formulas are proposed. The concepts of the virtual space and related mappings are proposed to achieve these objectives. The mapping between the logical space and the virtual space provides flexibility to avoid any cluster being bound to any specific location in a block. The mapping between the virtual space and the physical space provides further flexibility to avoid any block (and its LBAs) being bound to any physical block. A series of experiments is conducted to provide insights into configurations of the proposed method, compared with existing implementations. CNFTL could be implemented as a user module in the MTD subsystem or could be directly applied to many types

7. Flash-memory management is usually carried out by either software on a host system (e.g., that of SmartMedia and MemoryStick) or firmware of a flash device (e.g., that of CompactFlash and Disk on Module). Since hundreds of kilobytes in flash-memory management are often acceptable for software on a host system, the configurations of CNFTL-A or CNFTL-B can be adopted. When firmware is adopted for flash-memory management, configurations with smaller table storage than those of CNFTL-A or CNFTL-B must be set because there are only few kilobytes available on the corresponding devices.

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.

of flash memory, even with sequential write constraints such as *MLC* flash memory.

For future research, user access patterns for configurable FTL designs and implementations will be further explored. Hot data identification issues in configurable FTL designs will also be exploited. An integrated FTL design on garbage collection could be another research direction with good reward.

ACKNOWLEDGMENTS

This research was supported in part by research grants from the Taiwan, Republic of China National Science Council under Grant NSC95-2219-E-002-014, the Aim for Top University Program, and Genesys Logic, Inc.

REFERENCES

[1] Aleph One Company, Yet Another Flash Filing System, 2001.
[2] A. Ban, "Flash File System," US Patent 5,404,485, 1995.
[3] A. Ban and R. Hasharon, "Flash File System Optimized for Page-Mode Flash Technologies," US Patent 5,937,425, 1999.
[4] L.-P. Chang and T.-W. Kuo, "A Real-Time Garbage Collection Mechanism for Flash Memory Storage System in Embedded Systems," *Proc. Eighth Int'l Conf. Real-Time Computing Systems and Applications (RTCSA),* 2002.
[5] L.-P. Chang and T.-W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," *Proc. Eighth IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS '02),* pp. 187-196, 2002.
[6] L.-P. Chang and T.-W. Kuo, "An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems," *Proc. 19th Ann. ACM Symp. Applied Computing (SAC '04),* pp. 862-868, 2004.
[7] M.-L. Chiang and R.-C. Chang, "Cleaning Policies in Mobile Computers Using Flash Memory," *J. Systems and Software,* vol. 48, no. 3, pp. 213-231, 1999.
[8] 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.
[9] Compact Flash Assoc., CompactFlash™ 1.4 Specification, 1998.
[10] D. Woodhouse JFFS: The Journalling Flash File System, Red Hat, Inc., 2001.
[11] Intel Corp., "Understanding the Flash Translation Layer (FTL) Specification," 1998.
[12] H.J. Kim and S.G. Lee, "An Effective Flash Memory Manager for Reliable Flash Memory Space Management," *IEICE Trans. Information and Systems,* vol. E85-D, no. 6, pp. 950-964, 2002.
[13] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," *Proc. USENIX Winter Technical Conf.,* pp. 155-164, 1995.
[14] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for CompactFlash Systems," *IEEE Trans. Consumer Electronics,* vol. 48, no. 2, pp. 366-375, 2002.
[15] C. Park, J. Seo, S. Bae, H. Kim, S. Kim, and B. Kim, "A Low-Cost Memory Architecture with NAND XIP for Mobile Embedded Systems," *Proc. First IEEE/ACM/IFIP Int'l Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS '03),* pp. 138-143, 2003.
[16] SSFDC Forum, SmartMedia™ Specification, 1999.
[17] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile Main Memory Storage System," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '94),* pp. 86-97, 1994.

Jen-Wei Hsieh received the BS, MS, and PhD degrees in computer science and information engineering from National Taiwan University, in 1999, 2001, and 2006, respectively. He is currently an assistant professor in the Department of Computer Science and Information Engineering at National Taiwan University of Science and Technology. His current research interests include flash-memory storage systems and real-time systems. He is a member of the IEEE.

Yi-Lin Tsai received the BS degree in computer science from National Tsing-Hua University in 2003 and the MS degree in computer science and information engineering from National Taiwan University in 2005. He is currently an engineer at Realtek Semiconductor Corp., working on development of LCD-TV and DTV.

Tei-Wei Kuo received the BSE degree in computer science and information engineering from National Taiwan University, Taipei, in 1986 and the MS and PhD degrees in computer science from the University of Texas, Austin, in 1990 and 1994, respectively. He is currently a professor and the chairman of the Department of Computer Science and Information Engineering at National Taiwan University. Since February 2006, he has been a deputy dean at the same university. His research interests include embedded systems, real-time operating systems, and real-time database systems. He has more than 140 technical papers published or accepted in international journals and conferences. He has been an associate editor of the *Journal of Real-Time Systems* (since 1998) and the *IEEE Transactions on Industrial Informatics* (since 2007) and has been the Steering Committee Chair of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) since 2005. He also served as the program chair of the IEEE Real-Time Systems Symposium (RTSS) in 2007 and a program cochair of the IEEE Real-Time Technology and Applications Symposium (RTAS) in 2001. He has been an executive committee member of the IEEE Technical Committee on Real-Time Systems (TC-RTS) since 2005. He is a senior member of the IEEE.

Tzao-Lin Lee received the MS and PhD degrees in statistics from Carnegie Mellon University in 1981 and 1985, respectively. He is currently an associate professor in the Department of Computer Science and Information Engineering at National Taiwan University. His current interests include integrating database and XML applications for office automation, security, and privacy issues in server-client data storage and error correcting issues in write once streaming media.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: National Taiwan University. Downloaded on March 18, 2009 at 01:33 from IEEE Xplore. Restrictions apply.