JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 27, 1713-1727 (2011)

A Low-Memory Address Translation Mechanism for Flash-Memory Storage Systems*

CHIN-HSIEN WU, CHEN-KAI JAN+ AND TEI-WEI KUO++
Department of Electronic Engineering
National Taiwan University of Science and Technology
Taipei, 106 Taiwan
+*Department of Electrical Engineering*
Chang Gung University
Taoyuan, 333 Taiwan
++*Department of Computer Science and Information Engineering*
National Taiwan University
Taipei, 106 Taiwan

While flash-memory has been widely adopted for various embedded systems, the performance of address translation has become a critical issue for the design of flash translation layers. The aim of this paper is to improve the performance of existing designs by proposing a caching mechanism for efficient address translation. A replacement strategy with low-time complexity and low-memory requirements is proposed to cache the most recently used logical addresses. According to the experiments, the proposed method has shown its efficiency in the reducing of the address translation time.

Keywords: flash memory, caching mechanism, embedded systems, storage systems, low memory

1. INTRODUCTION

Flash translation layers can provide block-device emulation for NAND flash-memory. Many well-known file systems (*e.g.*, FAT, EXT2, and NTFS) can be easily and transparently built on the flash-memory devices without any modifications. Currently, there are two kinds of flash translation layers: FTL (Flash Translation Layer) [1-3, 7] and NFTL (NAND Flash Translation Layer) [8, 9]. The main problem of FTL is its fine-grained address translation design because of large main memory space for the address translation information. NFTL is proposed to resolve this problem by adopting a more coarse grain in address translation, but linear searches might be needed with each data access. The objective of this research is to improve the performance of existing designs by proposing a caching mechanism for efficient address translation. The goal is to improve the performance of coarse-grained flash translation layers with limited main memory space for caching. Data structures are proposed to accelerate the matching of a given logical address and its corresponding physical address on flash-memory. The most recently used mapping entries of logical addresses and their corresponding physical addresses are managed intelligently. With the NP-Completeness of the mapping-entry replacement problem, we pro-

Received November 11, 2009; revised February 23 & May 24 & August 16, 2010; accepted September 23, 2010. Communicated by Chung-Ta King.
* This paper was an extended version of a paper appeared in the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, April 24-26, 2006, Kolon Hotel, Gyeongju, Korea, and was partially supported by the National Science Council of Taiwan, R.O.C., under Grant No. NSC 100-2628-E-011-014-MY3-.

pose two weight-based replacement algorithms to select proper mapping entries for replacement. We also implement a LRU (Least Recently Used) caching mechanism for comparison. The experiments are run under a realistic workload trace and the experimental results show that the proposed approach can reduce the address translation time by 10% to 30% with only main memory space from 16KB to 512KB when a 20GB storage device is used.

The rest of this paper is organized as follows: Section 2 provides the overview of flash-memory. Section 3 is the related work. Section 4 provides the motivation. Section 5 introduces an address translation caching mechanism. Section 6 provides the performance evaluation. Section 7 is the conclusion.

2. OVERVIEW OF FLASH-MEMORY

A NAND flash-memory chip consists of multiple blocks, and each block is made up of a fixed number of pages. A block is the smallest unit that erase operations can be applied to, while read and write operations can be done in pages. A page contains a user area and a spare area, where the user area is reserved for the storage of a logical block and the spare area stores ECC and other house-keeping information, such as the logical block address (LBA). Note that an LBA denotes a logical address of a read/write unit in the flash-memory. The typical size of the user area and spare area of a page is 512B and 16B, respectively. The typical block size of a NAND flash-memory chip is 16KB. Flash-memory is write-once, therefore we do not overwrite existing data on each update. Instead, data must be written to free space and old versions of the data are invalidated, or considered as dead. This update strategy is known as "out-place update," meaning that any existing data on flash-memory cannot be over-written (updated) unless it is first erased. The pages that store live data are called "valid (live) pages" and ones containing dead data are referred to as "invalid (dead) pages". After performing several write operations, the amount of free space on flash-memory may be quite low. Activities (that consist of a series of read, write, and erase operations) are executed to recycle invalid pages. These activities are known as "garbage collection" and are considered as overhead in flash-memory management.

Since flash-memory is write-once, data can not be overwritten. Instead, data are written to free space, and the old versions of data are invalidated (or considered as dead). In order to resolve the residing location problem for data on flash-memory, a flash translation layer is proposed to emulate flash-memory as block devices so that many existing file systems (*i.e.*, FAT/DOS, EXT/EXT2, and NTFS, *etc.*) can be built on them without any modifications. Usually, a flash translation layer contains a RAM-resident translation table, and each entry of the table contains the corresponding physical address for a logical address.

3. RELATED WORK

3.1 Flash Translation Layer

For transparently accessing flash-memory, the block-device emulation approach results in popular deployments of flash-memory technology. Devices with the block-device

emulation approach are CompactFlash [4], DiskOnChip [5], and SmartMedia [6]. Recently, many manufacturers (such as SanDisk, STEC, Samsung, Mtron, and PNY) have released a huge-capacity NAND-based SSD (Solid-State Device) such as a 64GB SSD in the markets. A NAND-based SSD is widely used in PC-based and embedded computing systems. Many well-known file systems can be used with such flash-memory devices. A NAND-based SSD consists of raw NAND flash-memory chips and a controller. The controller handles data signals, control signals, and address signals from the host system and, at the same time, provides block-device emulation (*i.e.*, FTL and NFTL) over NAND flash-memory chips. A NAND-based SSD has a controller which is equipped with RAM and ROM. RAM is used for storing firmware execution code and temporary data. ROM stores basic routines (*e.g.*, read, write, and erase operations) for accessing NAND flash-memory and main code for providing block-device emulation. Due to the hardware characteristics of flash-memory, the controller can access flash-memory and RAM simultaneously. Note that DiskOnChip and CompactFlash have the similar architecture to a NAND-based SSD.

FTL and NFTL are two popular types of flash translation layers. FTL adopts a page-level (*i.e.*, fine-grained) address translation. The main problem of FTL is its large main memory space for storing the address translation information. As a result, NFTL is proposed for the huge-capacity flash-memory storage systems since NFTL adopts a block-level (*i.e.*, coarse-grained) address translation. Currently, there are any famous flash translation layers such as AFTL [10], BAST [11], FAST [11], and N + K [12] for performance improvement. They also adopt a small fine-grained translation table for improving the address translation. Since the table uses a LRU-based replacement mechanism, we want to compare the proposed method with the LRU-based replacement mechanism in the paper.

3.2 TLB

We must point out that the objective of the paper is close to a kind of cache memory: TLB (Translation Look-aside Buffer) [14, 15]. Distinct from the past work on cache memory [13-17], the proposed method in this paper targets flash-memory. Since TLB is a hardware cache (*i.e.*, SRAM), its manipulation and replacement should be simple for efficiency. Currently, there are two modern microprocessors: the Intel Nehalem and the AMD Opteron X4 [14]. The TLB organization of the Intel Nehalem adopts a four-way set associative cache with a LRU replacement. For the AMD Opteron X4, its TLB organization adopts a four-way or fully associative cache with a LRU replacement. TLB consists of entries and an entry will contain a tag (*i.e.*, a virtual page address), a physical page address, and some bits for maintenance (*i.e.*, valid bits, dirty bits, and reference bits). Since a typical replacement method in TLB is a kind of the LRU-based method, reference bits are increased when the corresponding entries are accessed. An entry with the smallest count will be replaced out when TLB is full.

When we compare flash-memory with TLB, the access time of flash-memory and TLB is about 60us-2ms and 0.5-2.5ns, respectively. Therefore, flash-memory is a slower device such that an address translation mechanism in flash-memory can have more design flexibility. That is, a sophisticated data structure and a replacement strategy can be executed by software for efficient address translation. Furthermore, TLB adopts a counter-based method (*i.e*, reference bits) to emulate a LRU-based method since a real LRU link-list is not suitable for hardware design but its performance is better than the counter-based

method. That is why we implement a real LRU link-list and compare with the proposed method.

4. MOTIVATION

NFTL represents a typical coarse-grained flash translation layer. An LBA under NFTL is divided into a virtual block address and a block offset, where the virtual block address (VBA) is the quotient of the division of the LBA by the number of pages in a block, and the block offset is the remainder of the division. Each VBA is associated with a primary block and a replacement block. When a write request is issued, the content of the write request is written to the page with the corresponding block offset in the primary block. Note that a write request in the paper is defined as a write to an LBA. Since flash-memory is write-once, any subsequent write of the same LBA is written to the first free page in the corresponding replacement block. NFTL must maintain a table in which each entry has a primary block address and a replacement block address. When a read request is issued, NFTL must locate the most-recent content by searching the primary block and the replacement block whenever necessary. If the replacement block exists, NFTL must read every spare area in the replacement block to find the most-recent content backwardly from the end of the replacement block since NFTL sequentially writes data in the replacement block. If the most-recent content can not be found in the replacement block, then NFTL locates the page in the primary block by the primary block address and the block offset. Note that the read time of a spare area of flash-memory is about 50us [18].

Such an observation motivates the goal of this research. That is how to improve the existing flash translation layers by our proposed method. The address translation problem can be more serious when NFTL adopts a more coarse-grained address translation than a block-level address translation. Although NFTL is the target flash translation layer in the paper, we must point out that the proposed method can be used for other existing coarse-grained flash translation layers such as AFTL, BAST, FAST, and N + K.

5. AN ADDRESS TRANSLATION CACHING MECHANISM

5.1 Overview

We will propose a low-memory address translation caching mechanism (LMT) for efficient address translation, as shown in Fig. 1. The definition of low memory means that LMT can provide en efficient address translation mechanism when the available main memory space is small. The purpose of LMT is to accelerate the translation of a given logical address to its corresponding physical address. Since we use a B-tree-like mechanism to organize LMT, LMT resembles a balance-tree data structure in insertions, deletions, and reorganization. LMT consists of translation nodes, where each translation node contains pointers to child translation nodes for tree traversal. The least recently used (LRU) link lists are attached to each leaf translation node, and each item in the link lists is a translation unit. Each translation unit maps a range of logical addresses to a continuous space of physical addresses.

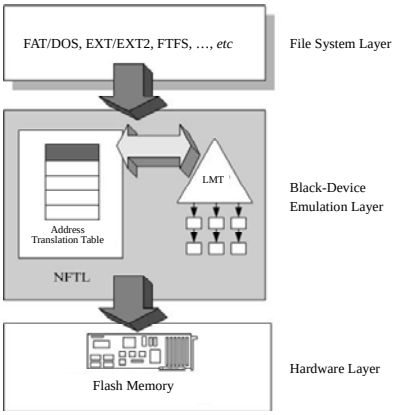


Fig. 1. System architecture.

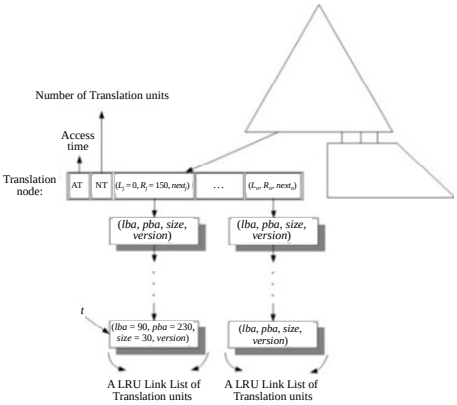


Fig. 2. An example of LMT.

5.2 Data Structures: Translation Node/Translation Unit

Two data structures are proposed in this section, as shown in Fig. 2: *translation node* and *translation unit*. The idea is to manage the mapping of logical addresses and physical addresses in terms of a hierarchical structure of address ranges. Assume that *FANOUT* is the maximum fan-out of a translation node in the proposed mechanism, where the performance of the proposed mechanism would be evaluated with different fan-outs in the experiments. Each translation node is defined as a tuple (*AT*, *NT*, (L_1, R_1, $next_1$), …, (L_i, R_i, $next_i$)) for $0 < i \leq FANOUT$, where *AT* and *NT* denote the access time of the translation node and the number of the translation units under the translation node, respectively. Note that a system counter is used to denote the latest access time in this paper. When each operation (*i.e.*, insert or search operations) on LMT is finished, the system counter is increased one. (L_i, R_i, $next_i$) points to a child translation node whose bounding range is [L_i, R_i], where $L_i < R_i$. For any (L_i, R_i, $next_i$) and (L_j, R_j, $next_j$), $R_i < L_j$ if $0 < i < j \leq FANOUT$. Note that if the translation node is a leaf node, $next_i$ will point to a (LRU) link list of translation units whose bounding range is [L_i, R_i]. The translation units in the (LRU) link list are ordered by the access sequence (*i.e.*, the most recently used translation unit is changed to the head of the link list). Each translation unit is defined as a tuple (*lba*, *pba*, *size*, *version*). *lba* and *pba* denote the starting logical address and the starting physical address of a continuous space on flash-memory, respectively. Since the sequential updates of write requests might be stored in a continuous space on flash-memory, *size* denotes the range size of the continuous space on flash-memory. *version* is to reflect the recency of the translation unit. When a new translation unit is created, its version tag is assigned as a system counter value and then the counter is increased one. Therefore, we can compare the version tags to understand their recency orders. As shown in Fig. 2, assume that a read request whose LBA = 100 is issued and a corresponding leaf translation node whose bounding range contains LBA = 100 will be traversed. A translation unit t = (*lba* = 90, *pba* = 230, *size* = 30, *version* = 3) will be retrieved from a link list in the leaf translation node. Therefore, the corresponding physical address can be calculated and is (100 – *t.lba* + *t.pba*) = (100 – 90 + 230) = 240.

5.3 Manipulations of Translation Node/Translation Unit

5.3.1 Translation node

Whenever a write request is issued, a new translation unit that describes the logical address and the corresponding physical address will be created and inserted into LMT. LMT will be traversed from the root to reach a proper leaf translation node for the insertion. Let t be the new translation unit and ($t.lba$, $t.pba$, $t.size$, $t.version$) be the starting logical address, the starting physical address, the range size, and the version, respectively. Let $t.LR$ denote the range of continuous logical addresses in the interval [$t.lba$, $t.lba$ + $t.size$). Assume that a translation node N, which denotes (AT, NT, (L_1, R_1, $next_1$), …, (L_i, R_i, $next_i$)) for $0 < i \leq FANOUT$, is traversed for the insertion of t into LMT. Let Θ be the set such that $\forall(L_m, R_m, next_m) \in \Theta$ overlap with $t.LR$ for $0 < m \leq FANOUT$. For each (L_m, R_m, $next_m$), there could be three cases about the processing of t and (L_m, R_m, $next_m$) in the following:

Case 1: If $[L_m, R_m] \subseteq t.LR$, then all translation nodes and translation units in the subtree rooted by $next_m$ in LMT are removed. Finally, (L_m, R_m, $next_m$) is also removed.

Case 2: If $[L_m, R_m]$ covers $t.LR$ completely, then t is traversed recursively into the subtree rooted by $next_m$ for the insertion.

Case 3 (a): Choose a (L_m, R_m, $next_m$) that can have a minimum enlargement to include ($[L_m, R_m] \cup t.LR$) and then $[L_m, R_m]$ in (L_m, R_m, $next_m$) is changed to the new range ($[L_m, R_m] \cup t.LR$). Then, t is traversed recursively into the subtree rooted by $next_m$ for the insertion.

Case 3 (b): On the other hand, $\forall(L_n, R_n, next_n) \in \Theta - (L_m, R_m, next_m)$, if $t.LR \cap [L_n, R_n] \neq \phi$, $[L_n, R_n]$ in (L_n, R_n, $next_n$) is changed to the new range ($[L_n, R_n]$) − ($[L_n, R_n] \cap t.LR$). All intervals of translation nodes and translation units in the subtree rooted by $next_n$ are changed accordingly.

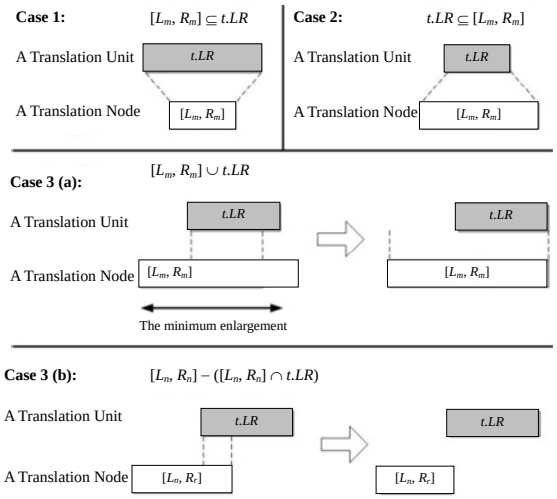


Fig. 3. Three cases about the processing of t and a translation node N.

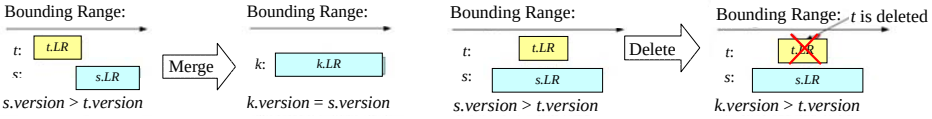
After the processing of t and all (L_m, R_m, $next_m$), t would be traversed recursively into the next translation node for the insertion. In Cases 2 and 3, there exists one (L_m, R_m, $next_m$) $\in \Theta$ and t can be traversed recursively into the subtree rooted by $next_m$ for the insertion. When the fan-out of a translation node N is full, the translation node must be split into two half parts, and AT of the two split translation nodes is set as the latest system counter. NT and $version$ of the two split translation nodes are also changed accordingly. On the other hand, if the fan-out of a translation node N is less than a half of $FANOUT$ during the insertion, we propose not to merge the translation node with others for simple implementation. We remove a translation node from LMT when it becomes empty.

5.3.2 Translation unit

When a translation unit is traversed to a leaf translation node, the translation unit will be inserted into a link list of translation units in the leaf translation node. We propose two operations to reduce the length of the link list for the insertion, as follows: (Note that the time complexity of each merge/delete operation is $O(n)$, where n is the length of the link list.)

Merge: Let t be an existing translation unit in a link list, and s be a new translation unit for insertion. If ($s.lba$ + $s.size$) = $t.lba$ and ($s.pba$ + $s.size$) = $t.pba$, then t and s are merged into a new translation unit k such that k is equal to s, except that $k.size$ = ($s.size$ + $t.size$) and $k.version$ is redefined as needed. Similarly, if ($t.lba$ + $t.size$) = $s.lba$ and ($t.pba$ + $t.size$) = $s.pba$, then t and s are merged into a new translation unit j such that j is equal to t, except that $j.size$ = ($s.size$ + $t.size$) and $j.version$ is redefined as needed.

Delete: Assume that a new translation unit has been inserted. Let t be an existing translation unit in the link list, and S be the set of all translation units in the link list such that $\forall s \in S$, $s.version > t.version$ and $s.LR \cap t.LR \neq \phi$. If $t.LR \subseteq \cup_{s \in S}(s.LR)$, then t is removed from the link list.



(a) Merge. (b) Delete.

Fig. 4. Examples of merge and delete.

As shown in Fig. 4 (a), s and t can be merged to a new translation unit k. In Fig. 4 (b), t is deleted because of the insertion of s. When a read request is issued, the corresponding translation unit which contains the logical address of the read request will be searched from LMT.

5.4 A Replacement Strategy

The number of translation nodes and units for the most recently used logical addresses is constrained by available main memory space. If the main memory space used

by LMT is over a threshold, then some translation nodes and units must be replaced with new coming translation nodes and units. Two replacement algorithms are presented in the section. First, we define the replacement problem and show its NP-Completeness in the following:

Definition 1 The LMT replacement problem: Given a collection $T = \{N_1, N_2, N_3, \ldots, N_n\}$ of LMT translation nodes and two constants: S and C. $F_{size}(N_i)$ denotes the occupied main memory size by N_i, where there are m_i child translation nodes and k_i translation units in the subtree rooted by N_i. The memory size of a translation node and a translation unit is M_{node} and M_{unit}, respectively. Therefore, $F_{size}(N_i)$ is $m_i * M_{node} + k_i * M_{unit}$. $F_{weight}(N_i)$ denotes the weight of N_i and is the sum of all ATs of m_i child translation nodes. The problem is to find a subset T_s of T such that the total occupied main memory size of T_s is no less than S, and the total weight of T_s is no more than C.

Theorem 2 The LMT replacement problem is NP-Complete.

Proof: The LMT replacement problem can be reduced from the knapsack problem [21] directly based on the above definition. As a result, The LMT replacement problem is NP-Complete. ❑

Although dynamic-programming-based approximation algorithms [21] can derive good results for the knapsack problem, their time complexity might not be applicable to the LMT replacement problem in current embedded systems. Since the replacement problem is intractable, we will propose two heuristic replacement algorithms and, at the same time, to release sufficient free main memory space.

(1) A Weight-Based Replacement Algorithm

We propose a weight-based replacement algorithm to replace the least recently used part of LMT, and at the same time, to have more free main memory space. We design a weight function $W(N) = N.AT/N.NT$ where N is a leaf translation node. First, the replacement algorithm is to sort all leaf translation nodes by the weight function in an increasing order. The weight function is used for the replacement. We use a doubly linked list to maintain all leaf translation nodes in our implementation. When a replacement mechanism is required, all leaf translation nodes can be accessed and sorted by the doubly linked list. The time complexity is $O(n * \log n)$ for the sorting, where n is the number of all leaf translation nodes. The replacement algorithm is to release the leaf translation node with the minimum weight and will continue the replacement until sufficient free main memory space is reached. If a leaf translation node is chosen for the replacement, the translation units that belong to the node must be released and its parent node would be updated recursively.

(2) A Second-Chance Weight-Based Replacement Algorithm

We propose a second-chance weight-based replacement algorithm by revising the above algorithm. As shown in Fig. 5, since the beginning translation units of the LRU link lists of each leaf translation node are accessed more frequently than those translation units at the end of the link lists, we propose to give the beginning translation units a second

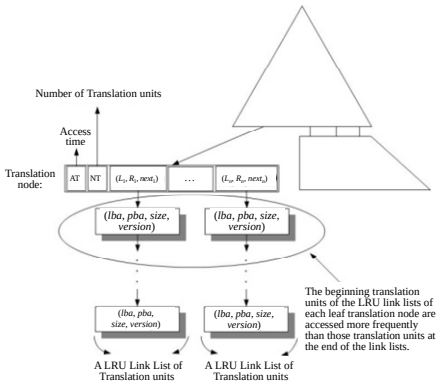
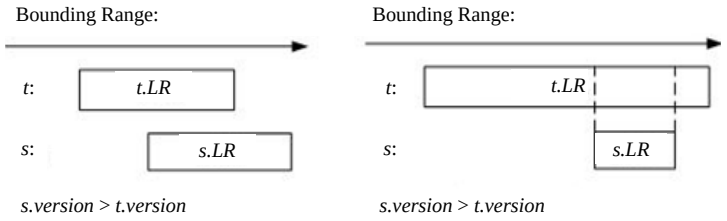


Fig. 5. A second chance for the beginning translation units.



(a) Hazard 1. (b) Hazard 2.

Fig. 6. Two potential hazards of the replacement strategy.

chance. However, as shown in Fig. 6, there could be two potential hazards if a straight replacement algorithm is implemented. Let *s* and *t* be two translation units. In Fig. 6 (a), assume that *s.version* > *t.version*, $s.LR \cap t.Lr \neq \phi$, and $s.LR \not\subset t.LR$. If *s* is picked up for replacement and *t* remains in LMT, then some mapping information (*i.e.*, the intersection of *t.LR* and *s.LR*) in *t* is out-of-date. Therefore, *t* is not a up-to-date translation unit because of *s.version* > *t.version* (referred to as Hazard 1). Fig. 6 (b) shows another potential hazard (referred to as Hazard 2) in address translation since the range of *s* is in that of *t*.

Two operations: prune and cut are proposed to resolve Hazards 1 and 2, respectively. Let Ψ and Ω denote the first half and the second half of one LRU link list of one leaf translation node *N* that will be replaced. Translation units in Ω would be replaced after prune and cut operations. After translation units in Ω are released, *N.AT* and *N.NT* of the leaf translation node *N* is then updated as *N.AT*/2 and *N.NT*/2, respectively.

Prune: Assume that $\forall t \in \Psi$ and $\forall s \in \Omega$, where *s.version* > *t.version* and $s.LR \cap t.LR \neq \phi$ and $s.LR \not\subset t.LR$, then the interval *t.LR* of *t* is modified as a new range $t.LR - (s.LR \cap t.LR)$. The prune operation is to resolve Hazard 1, as shown in Fig. 6 (a).

Cut: Assume that $\forall t \in \Psi$ and $\forall s \in \Omega$, where *s.version* > *t.version* and $s.LR \cap t.LR \neq \phi$ and $s.LR \subseteq t.LR$, then *t* will be split to two parts with bounding ranges [*t.lba*, *s.lba* − 1] and

[*s.lba* + *s.size* + 1, *t.lba* + *t.size*]. The smaller one would be discarded since the smaller one could have a lower opportunity to be used in the future. *t* should be updated to reflect the remaining larger part. The cut operation is to resolve Hazard 2, as shown in Fig. 6 (b).

As shown in Fig. 6 (b), *t* would be modified to describe the bounding range [*t.lba*, *s.lba* − 1] since [*t.lba*, *s.lba* − 1] is a larger part. A prune operation should be executed whenever one new translation unit is inserted. A cut operation should be executed whenever the replacement strategy is executed. The time complexity of a prune and a cut operation is $O(n^2)$ since Ψ and Ω will have at most $n/2$ translation units.

6. PERFORMANCE EVALUATION

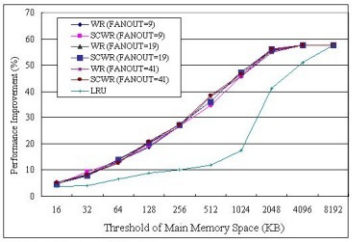
6.1 Experimental Setup and Performance Metrics

A 20GB NAND-based system prototype was built to evaluate the performance. The prototype was implemented with NFTL and ran the collected trace. The prototype was equipped with CPU (Intel Celeron D 350) and 2GB main memory. Note that the trace was collected during one week from a machine that was different from the prototype. In the trace, there were 13,198,805 and 2,797,996 sectors written and read, respectively, where each sector size was 512B. Note that a sector is a logical item which is viewed from the operating system and a page is a hardware unit in flash-memory. A sector is a read/write unit by the operating system and the logical address of a sector is called an LBA. In fact, the content of a sector is stored in a page. We must point out that there were 1,669,228 different LBA's that were accessed. The trace shows that many written data had spatial locality, where each LBA was written for 7.9 times on average. During the collection of the trace, real applications (such as Web Applications, E-mail Clients, MP3 Player, Media Player, Windows Office, MSN, Programming, and Virtual Memory Activities) were executed to have realistic and mixed workloads in daily life. Major I/O requests were issued by some specific applications such as Web Applications, E-mail Clients, and Virtual Memory Activities. In the nighttime, the number of I/O requests was smaller than that of daytime. Overall, most of I/O requests will be concentrated in a small part of the time. Each translation unit size was 20B and each translation node size was among 128B, 256B, and 512B, where *FANOUT* was 9, 19, and 41, respectively, in the experiments.

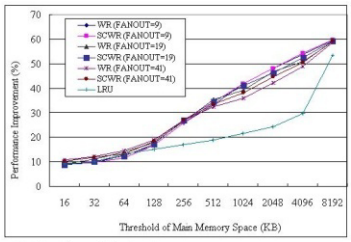
We abbreviate the weight-based replacement algorithm and the second-chance weight-based replacement algorithm to *WR* and *SCWR* hereafter. We also implemented a traditional *LRU-based* replacement algorithm (referred to as *LRU* hereafter). The *LRU* replacement algorithm can maintain slots in a hash table, where each slot contains the mapping information of a given logical address (*i.e.*, LBA) to its corresponding physical address. Note that the translation units of *WR* and *SCWR* are to map a range of logical addresses to a continuous space of physical addresses. When the occupied main memory size of *LRU* is beyond the threshold, the least recently used slots would be replaced with new slots. Each replacement algorithm (*i.e.*, *WR*, *SCWR*, and *LRU*) was run under different main memory space thresholds. A NFTL prototype was also implemented to be a comparison baseline. We compared *WR*, *SCWR*, and *LRU* with NFTL by measuring the performance of address translation time and the overhead under different main memory space thresholds.

6.2 Performance Improvement

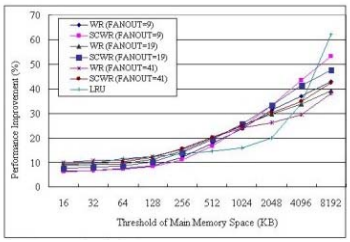
Since NFTL was a comparison baseline, we measured the performance improvement of address translation time by comparing *WR*, *SCWR*, and *LRU* with NFTL. The performance improvement of address translation time under different main memory space thresholds was shown in Figs. 7 (a)-(c). Note that two hours of the trace and one day of the trace are obtained from the trace of one week. We can observe that *WR*, *SCWR*, and *LRU* had better performance than *NFTL* in all cases over two hours, one day, and one week of the trace. The hit ratio of one week of the trace was shown in Table 1. If most address translations can be served by LMT, NFTL will not work as usual such that the performance of



(a) Two hours of the trace.



(b) One day of the trace.



(c) One week of the trace.

Fig. 7. Performance improvement.

Table 1. Hit ratio of one week of the trace.

Threshold of Main Memory Space (KB)	16	32	64	128	256	512	1,024	2,048	4,096	8,192
WR (FANOUT = 9)	0.121	0.126	0.143	0.173	0.219	0.278	0.342	0.401	0.524	0.603
SCWR (FANOUT = 9)	0.122	0.126	0.146	0.173	0.223	0.282	0.349	0.41	0.53	0.61
WR (FANOUT = 19)	0.162	0.17	0.182	0.22	0.262	0.298	0.335	0.465	0.51	0.577
SCWR (FANOUT = 19)	0.153	0.161	0.178	0.218	0.263	0.299	0.34	0.455	0.527	0.584
WR (FANOUT = 41)	0.202	0.213	0.224	0.241	0.268	0.308	0.41	0.452	0.482	0.562
SCWR (FANOUT = 41)	0.197	0.211	0.222	0.238	0.266	0.295	0.398	0.452	0.501	0.562
LRU	0.195	0.22	0.233	0.244	0.253	0.265	0.281	0.314	0.445	0.679

address translation can be improved. Therefore, we can observe that the hit ratio was proportional to the ratio of performance improvement. We can also observe that *WR* and *SCWR* were superior to *LRU* under many thresholds, especially when the main memory space was not large. Since *WR* and *SCWR* adopted the translation nodes and the translation units to map a range of logical addresses to a continuous space of physical addresses, main memory space can be exploited efficiently to store more mapping information. As a result, the performance improvement of address translation time can increase significantly with low main memory space. However, when the main memory space was larger (*i.e.*, > 8192KB), *LRU* could have equal or better performance than *WR* and *SCWR*. This was because the larger main memory space can favor *LRU* to keep more to-be-used translation units than *WR* and *SCWR*. We must point out that the aim of the proposed method is to provide efficient address translation with low main memory space such that *LRU* is not applicable to a low main memory environment.

When the main memory space was smaller (*e.g.*, < 512KB), *WR*/*SCWR* with bigger *FANOUT* had better performance than *WR*/*SCWR* with smaller *FANOUT*. This was because smaller *FANOUT* could result in more replacements of translation nodes than larger *FANOUT*. A lot of replacements of translation nodes could replace more to-be-used translation units under smaller main memory space such that *WR*/*SCWR* with bigger *FANOUT* had better performance. On the other hand, when the main memory space was larger (*e.g.*, > 512KB), *WR*/*SCWR* with smaller *FANOUT* had better performance than *WR*/*SCWR* with bigger *FANOUT*. This was because larger main memory space could let the least recently used translation nodes be replaced soon. A lot of replacements of the least recently used translation nodes could result in *WR*/*SCWR* with smaller *FANOUT* had better performance. The relationship between main memory space, FANOUT, and performance is shown in Table 2. We can also observe that when the main memory space was larger (*e.g.*, > 1,024KB), *SCWR* can perform better than *WR*, as shown in Fig. 7 (c). Since *SCWR* adopts the second chance, larger main memory space can favor *SCWR* to keep more to-be-used translation units. On the other hand, in most memory space thresholds of Fig. 7 (c), *SCWR* had the same performance with *WR*.

Table 2. Relationship between main memory space, FANOUT, and performance.

Main Memory Space	FANOUT	Performance
Small	Big	Better
Large	Small	Better

Overall, 256-512KB for *WR* and *SCWR* might be good enough to I improve the performance of NFTL (about 20-40%) for 20GB flash-memory devices. Furthermore, when the main memory space can be adjusted dynamically, *SCWR* could have better performance.

6.3 Overhead

The overhead of the search and insertion of one translation unit was shown in Table 3 under different main memory space thresholds and *FANOUT*s. When the main memory space or the translation node size was increased, the overhead was also increased.

Table 3. Overhead of average search time and average insertion time.

Threshold of Main Memory Space (KB)	16	32	64	128	256	512	1,024	2,048	4,096	8,192
WR (FANOUT = 9) Ave. Search Time (us)	1.36	1.39	1.44	1.53	1.54	1.59	1.59	1.7	2.1	4.45
Ave. Insertion Time (us)	6.31	8.52	9.42	10.53	10.7	11.71	15.22	18.27	20.4	34.05
SCWR (FANOUT = 9) Ave. Search Time (us)	1.31	1.36	1.42	1.49	1.52	1.94	1.61	1.76	1.84	5.1
Ave. Insertion Time (us)	6.44	8.15	9.11	10.61	10.94	12.75	23.71	31.85	38.94	52.77
WR (FANOUT = 19) Ave. Search Time (us)	1.33	1.43	1.53	1.54	1.56	1.64	1.66	1.75	5.15	5.47
Ave. Insertion Time (us)	7.41	9.13	10.49	10.57	10.73	10.84	10.95	11.43	32.22	35.76
SCWR (FANOUT = 19) Ave. Search Time (us)	1.38	1.42	1.47	1.5	1.6	1.6	1.67	1.74	4	6.15
Ave. Insertion Time (us)	7.59	9.52	10.49	10.77	10.99	11.09	11.75	13.66	28.16	41.4
WR (FANOUT = 41) Ave. Search Time (us)	1.45	1.53	1.58	1.53	1.58	1.6	1.67	3.79	3.65	5.29
Ave. Insertion Time (us)	9.95	10.23	10.57	10.62	10.71	10.95	11.07	23.83	23.29	34.81
SCWR (FANOUT = 41) Ave. Search Time (us)	1.44	1.51	1.51	1.54	1.57	1.28	1.67	1.87	2.6	6.07
Ave. Insertion Time (us)	9.96	10.26	10.64	10.71	10.87	11.1	11.6	13.12	17.28	38.94
LRU Ave. Search Time (us)	1.16	1.13	1.12	1.05	1.19	1.51	2.34	3.4	4.65	2.4
Ave. Insertion Time (us)	3.29	3.27	4.45	3.94	4.77	5.18	6.97	10.5	15.51	16.98

The average insertion time of one translation unit for *WR* and *SCWR* could be longer that of *LRU* since the manipulations of the translation nodes and the translation units needed more time. Note that the search time of one translation unit for *WR* and *SCWR* were very close to that of *LRU*. For example, assume that a read request is issued and its translation unit can not be retrieved from LMT, the search time of a translation unit is about 1.6us for *WR* and *SCWR* under 512KB main memory space. On the other hand, when a new translation unit would be inserted to LMT, the insertion of the translation unit can be done together with the programming of a page on flash-memory. This was because the insertion time of a translation unit (*i.e.*, 6-50 us) is less than the write time of a page (*i.e.*, 250 us). As a result, the manipulations of the translation nodes and the translation units for *WR* and *SCWR* can be overlapped with the read time of a page (*i.e.*, 50us) and the write time of a page (*i.e.*, 250us) [18], respectively.

7. CONCLUSION

This paper proposes an address translation caching mechanism for efficient address translation. A balance-tree-like data structure and manipulations are proposed to manage the address mapping between the most recently used logical addresses and their corresponding physical addresses. Two replacement algorithms are also presented to intelligently exploit the main memory space for caching the most recently used mapping information. The proposed method was evaluated by a NAND-based prototype and realistic workloads. It was shown that the system performance can be significantly improved with low main memory space. For example, the proposed method only needs about 512KB

main memory to have about 40% performance improvement. Furthermore, the overhead of the proposed method was also evaluated, and the results were very encouraging.

For future research, we should further explore efficient and hybrid replacement strategies for flash-memory address translation, especially designed for different embedded applications. With joint considerations of application designs and flash-memory characteristics, much more performance improvement could be achieved with less system overhead.

REFERENCES

1. Intel Corporation, "Understanding the flash translation layer (FTL) specification," 1998.
2. Intel Corporation, "Software concerns of implementing a resident flash disk," 1998.
3. Intel Corporation, "FTL logger exchanging data with FTL systems," 1998.
4. Compact Flash Association, "Compact flash 1.4 specification," 1998.
5. http://www.m-systems.com/site/en-US/Products/DiskOnChip/DiskOnChip.
6. SSFDC Forum, "Smart media specification," 1999.
7. "Flash file system," US Patent No. 5,404,485, 1995.
8. "Flash file system optimized for page-mode flash technologies," US Patent No. 5,937,425, 1999.
9. J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compact-flash systems," *IEEE Transactions on Consumer Electronics*, Vol. 48, 2002, pp. 366-375.
10. C. H. Wu, H. H. Lin, and T. W. Kuo, "An adaptive flash translation layer for high-performance storage systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 29, 2010, pp. 953-965.
11. S. W. Lee, W. K. Choi, and D. J. Park, "FAST: An efficient flash translation layer for flash memory," *ACM Transactions on Embedded Computing Systems*, Vol. 6, Article 18, 2007, pp. 879-887.
12. C. Park, W. Cheon, Y. Lee, M. S. Jung, W. Cho, and H. Yoon, "A re-configurable FTL (flash translation layer) architecture for NAND flash based applications," in *Proceedings of IEEE International Workshop on Rapid System Prototyping*, 2007, pp. 202-208.
13. A. J. Smith, "Cache memories," *ACM Computing Surveys*, Vol. 14, 1982, pp. 473-530.
14. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, Morgan Kaufmann Publishers, San Francisco, 2009.
15. A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed., John Wiley and Sons, Inc., New Jersey, 2009.
16. J. B. Chen, A. Borg, and N. P. Jouppi, "A simulation-based study of TLB performance," in *Proceedings of the 18th International Symposium on Computer Architecture*, 1991, pp. 114-123.
17. J. Choi, J. Lee, S. Jeong, S. Kim, and C. Weems. "A low power TLB structure for embedded systems," *IEEE Computer Society Technical Committee on Computer Architecture*, Vol. 1, 2002, pp. 3-3.

18. Samsung Electronics, NAND flash-memory datasheet and SmartMedia data book, 2008.
19. http://www.samsung.com/Products/Semiconductor/Flash/index.htm.
20. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co. Ltd., New York, 1979.
21. V. V. Vazirani, *Approximation Algorithm*, Springer Publisher, New York, 2001.

Chin-Hsien Wu (吳晉賢) received his B.S. degree in Computer Science from National Chung Cheng University in 1999. He received his M.S. and Ph.D. degree in Computer Science from National Taiwan University in 2001 and 2006, respectively. Now, he is an Assistant Professor at the Department of Electronic Engineering in National Taiwan University of Science and Technology. He is also a member of ACM and IEEE. His research interests include embedded systems, real-time systems, ubiquitous computing, and flash-memory storage systems.

Chen-Kai Jan (詹程凱) received his B.S. degree in Electrical Engineering from Chang Gung University, Taiwan, in 2008. He is currently working towards the Ph.D. degree in Electrical Engineering at Chang Gung University, Taiwan, R.O.C. His current research interests include medical imaging and therapeutic ultrasound.

Tei-Wei Kuo (郭大維) received the B.S.E. degree in Computer Science and Information Engineering from National Taiwan University in Taipei, Taiwan, in 1986. He received the M.S. and Ph.D. degrees in Computer Sciences from the University of Texas at Austin in 1990 and 1994, respectively. He is currently a Professor and the Chairman at the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, R.O.C. His research interests include embedded systems, real-time operating systems, and real-time database systems.