LS-LRU: A Lazy-Split LRU Buffer Replacement Policy for Flash-Based B^+-tree Index*

RIZE JIN, HYUNG-JU CHO, AND TAE-SUN CHUNG
Department of Computer Engineering
Ajou University
Suwon 443-749, Republic of Korea
E-mail: {jinrize, hjcho, tschung}@ajou.ac.kr

Most embedded systems are equipped with flash memory owing to its shock resistance, fast access, and low power consumption. However, some of its distinguishing characteristics, including out-of-place updates, an asymmetric read/write/erase speed, and a limited number of write/erase cycles, make it necessary to reconsider the existing system designs to explore its performance potential. For example, the buffer replacement policy of flash-based systems should not only consider the cache hit ratio, but also the relative heavy write and erase costs that are caused by flushing dirty pages. Most of the recent studies on buffer designs have focused on a Clean-First LRU strategy that evicts clean pages preferentially to reduce the number of writes to flash memory. However, each of them fails to distinguish dirty pages, which may have a different effect on the flash memory. In this paper, we propose a Lazy-Split LRU-based buffer management scheme that not only considers an imbalance of the read and write speeds but also different effects of different dirty pages and frequent changes of the B^+-tree index structure caused by intensive overwrites. Specifically, it introduces a semi-clean state to further classify some dirty pages into clean part and dirty part and several efficient replacement policies to reduce the number of B^+-tree splits. The experimental results show that our solution outperforms other algorithms including pure LRU and CFDC, and is effective and efficient for improving the performance of B^+-tree on flash memory.

Keywords: buffer management; B^+-tree; flash memory; replacement policy; split policy

1. INTRODUCTION

Flash memory [1] has many advantages, such as a smaller size, lower power consumption, shock resistance, and fast random access speed. It is therefore used in a wide spectrum of computing devices, such as embedded sensors, digital cameras, mobile phones, PDAs, and even personal computers and enterprise servers. However, compared to traditional storage devices, flash memory possesses many distinguishing electronic limitations: (1) erase-before-write, (2) an asymmetric read (80μs)/write (200μs)/erase (1.5ms) speed, and (3) a limited number of write/erase cycles. These limitations make the existing systems perform poorly with flash-based storage [2-4]. It is therefore necessary to reconsider the various system optimization issues. One of the major parts of a database design that need to be revisited is the buffer design, and in particular, a buffer replacement policy for a flash-based B^+-tree index, as B^+-tree frequently rewrites its

Received September 16, 2013; revised December 12, 2013; accepted xxx.
Communicated by Xiaodong Zhang.
* This work was partially supported by Basic Science Research Programs through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (NRF-2013R1A1A2A10012956 and NRF-2012R1A1A2043422).

existing nodes. Moreover, it becomes complicated under special circumstances, such as splitting, merging, and rotating, and it can propagate all the way to the root node.

Most recent studies on a flash-aware buffer design originate from the Clean-First LRU (CFLRU) replacement algorithm [5]. The basic idea behind this algorithm is to divide a Least Recently Used (LRU) list into a working region and clean-first region, and then, evict clean pages preferentially in the clean-first region. The new buffer replacement policies consider the cache hit ratio, as well as the replacement cost caused by flushing dirty victims. Note that the replacement cost of dirty pages is much larger than that of clean pages in terms of time and energy consumption.

We argue that these flash-aware buffer schemes are unsuitable for the implementation of B^+-tree indices in flash memory. Because they focus on only the asymmetric read/write cost, they cannot efficiently deal with frequent changes of B^+-tree caused by the arbitrary eviction of dirty nodes. The introduction of intensive overwrites can cause a large number of time-consuming write/erase operations for flash memory, which can be a bottleneck of database systems. This problem motivated us to design a novel buffer management scheme, Lazy-Split LRU (LS-LRU), for a flash-based B^+-tree index. The key features of LS-LRU are as follows:

- LS-LRU maintains buffered index entries in a B^+-tree structure. This has a twofold significance: index entries can be efficiently maintained as the buffer size increases, and as the entries are kept in a sorted order, we can perform a partial flush. This affects only a certain part of the base index stored in flash memory.
- LS-LRU introduces a semi-clean concept to make a more accurate classification for buffered dirty nodes.
- By introducing a family of efficient policies, LS-LRU reduces the amount of reorganization of B^+-tree and limits obsessive modifications of the memory buffer.

LS-LRU can significantly reduce the number of write and erase operations as well prevent degradation of the buffer hit ratio. Experimental results show that LS-LRU markedly outperforms its competitors under various traces in terms of execution time and energy consumption.

The rest of this paper is organized as follows. Section 2 presents the characteristics of flash memory and the existing buffer replacement polices. In Section 3, LS-LRU is described. Section 4 reports a performance simulation with several workload traces. Finally, we offer some concluding remarks regarding this research in section 5.

2. BACKGROUND

In this section, we detail the most important properties of flash memory, and review some related work that has motivated us to devise a new buffer management scheme.

2.1 Flash Memory

Flash memory [1], a magnetic, non-volatile storage device, is organized into a certain number of blocks, where each block consists of multiple pages. The size of a page

is typically 512, 2,048, or 4,096 bytes, with a few bytes (typically 16) of each page being used for storing an error detection and correction checksum. Table 1 shows a comparison of the access feature of 32 M × 8 bit flash memory and a magnetic disk.

Table 1: Access features [1]

Flash Memory	Page Read	Random Access	80 μs (Max.)
		Serial Page Access	50 ns (Min.)
	Page Write	Program Time	200 μs (Typ.)
		Block Erase Time	1.5 ms (Typ.)

The internals of flash memory differ in almost every aspect from those of a magnetic disk. A magnetic disk is a mechanical rotating storage device, whereas flash memory can be accessed and reprogrammed electronically. Flash memory therefore has many advantages over a magnetic device, such as a smaller size, lower power consumption, shock resistance, and a fast random access speed, which make it one of the most promising types of nonvolatile storage media. On the other hand, unlike a magnetic device that performs read and write operations at an equal speed, the write and erase operations of flash memory take much longer than a read operation. Reads and writes are performed at the page unit, whereas erasures can be performed only at the block unit. In addition, flash memory has no overwrite ability. An update must therefore be performed on a used page after erasing the corresponding block. Another limitation is that flash memory has a limited number of erase cycles per block, commonly between 100,000 and 1,000,000 [2].

2.2 Related Work

When the buffer is full and the current requested page is not in the buffer, the replacement policy has to select an in-buffer page as a victim. If a victim is a dirty page, it will be written back to the disk before paging in the requested page to guarantee data consistency.

Most operating systems [4-7] use an approximation of an LRU algorithm for a replacement policy customized for a magnetic-disk based storage system. LRU assumes that there is no difference in the time delay between read and write operations. In general, LRU stores some portion of the disk blocks in Random Access Memory (RAM) to reduce the number of I/O requests. Its goal is to minimize the buffer miss ratio for a given buffer size. However, a flash disk usually demonstrates extremely fast random read speeds, but relatively slow random write speeds. Operating systems with flash memory should therefore adopt a new replacement policy that considers the cache hit rate as well as the replacement cost.

During the past few years, many studies have emerged on the design of a buffer management scheme for flash memory. Wu *et al.* [8] proposed BFTL, which is a software module for efficiently using a B-tree index in flash memory. BFTL consists of a reservation buffer and a node translation table. The reservation buffer constructs an index unit to reflect request operations. When full, the cached index entries are sequentially written onto a single page even if they belong to different nodes. The node translation table is used to keep this kind of mapping information between the logical nodes and

physical pages. In this manner, BFTL succeeds in a reduction of the write operation count, but has a great increase in the search time. In addition, the first-fit commit policy of BFTL is too naive to consider for the buffer hit ratio.

Park *et al.* [5] proposed a new buffer replacement algorithm called Clean-First LRU (CFLRU) that considers the imbalance between the read and write costs of flash memory while replacing pages. The basic idea behind CFLRU is to deliberately keep a certain amount of dirty pages in the buffer pool to reduce the number of write operations to the flash memory, while preventing the overall performance from being significantly affected from the degradation of the buffer hit ratio. Specifically, CFLRU divides the LRU list into a working region and clean-first region, as shown in Fig. 1. The working region consists of recently used pages, and most of the cache hits are generated in this region. The clean-first region consists of candidate pages for eviction. CFLRU selects a clean page to evict in the clean-first region first to save flash write costs. If there is no clean page in this region, a dirty page at the end of the LRU list is evicted.

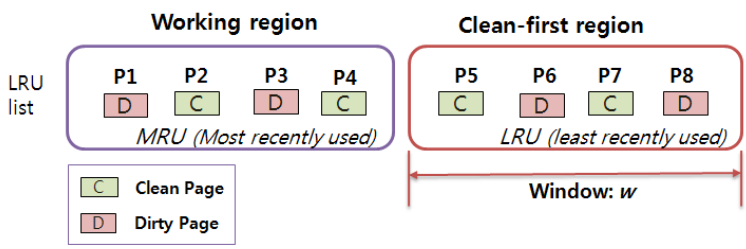


Fig. 1: CFLRU buffer management scheme [5].

For instance, suppose that pages were recently accessed in the order of P8, P7, P6, and P5 (such that P7 is the most recently used clean page, and P8 is the least recently used dirty page). Under the LRU page replacement algorithm, the last page (P8) is preferentially evicted. However, when using flash memory it may be beneficial to first evict the clean page, P7, to reduce the number of flash writes, even though this page was accessed more recently than the dirty page, P8. Because the buffer miss ratio may increase if a recently used clean page is evicted, it is important to adjust the size of the clean-first region (called the window size, *w*) properly. Here, only clean pages within the clean-first region can be candidate victims. If there are no clean or dirty pages within the clean-first region, then it acts as a normal LRU algorithm, in which the least recently used page becomes a victim.

Based on the same idea, CFDC [6] makes further improvement on the efficiency of the write operations by clustering together dirty pages whose page numbers are close to each other. A CFDC cluster has a variable size, determined by the set of pages currently being kept, which is different from a block-level LRU list. Further, ACR [7] improves the overall hit ratio over CFLRU and CFDC by differentiating between the frequently requested dirty pages and once-requested dirty pages. DABC-NV [9] presented a new buffer manager that uses both multi-level cell (MLC) and single-level cell (SLC) flash memory chips. Furthermore, DABC-NV considers their heterogeneous features together, and thus allocates cache space based on the characteristics of both mediums. A recent work, long sequential accessed pages first (LSF) [10], tries to keep frequent accessed

pages (e.g., metadata pages) in the buffer as long as possible to improve the hit ratio. However, it makes no use of physical flash log information, and thus its effect is limited to flash-based systems. Seo *et al.* [11] and Liao *et al.* [12] made a further contribution to solving the log block thrashing and high block associativity problems of BAST and FAST [3]. When replacing the buffer pages, there methods consider the usage information of the log blocks.

3. A NOVEL BUFFER MANAGEMENT SCHEME

As mentioned earlier, there are two essential performance characteristics for implementing buffer management schemes on flash-based systems: 1) a fast random read speed but slow random write speed, and 2) an increase in the write throughput along with the size of the writes, as there exists some per-write latency that causes small writes to be slower than larger writes [4, 13].

The main idea behind our solution is to avoid the high cost involved in the update of a flash-based B^+-tree index using a three-state buffer management, clean-first replacement, and a lazy-split strategy. In detail, first, our buffer design uses a three-state scheme to make a more accurate classification for the buffering nodes and utilize space more effectively; in addition to clean and dirty states, a semi-clean state is introduced to further classify some dirty pages into clean and dirty parts. This contributes to an improvement in the buffer space utilization, and further reduces the number of writes to flash memory. Second, while replacing a page, an enhanced clean-first strategy is used; clean state pages or clean parts of semi-clean state pages are preferentially selected as victims over dirty state pages in the cold region. This helps reduce the write and erase counts of the blocks in flash memory, and lower the data migration overhead. Finally, LS-LRU allows the size of the in-memory logical node to grow to k ($k > 1$) times that of the on-flash physical node. Furthermore, when it has to evict the dirty buffer units, it flushes those that may cause fewer node splits. This contributes to a reduction in the amount of reorganizations of the B^+-tree. Next, we illustrate the data structure, and then describe the three-state buffer management scheme and a series of new buffer replacement algorithms, which are the main contributions of this paper.

3.1 Data Structures

3.1.1 B^+-buffer

We organized the whole architecture as shown in Fig. 2. Table 2 summarizes the notations used in this paper.

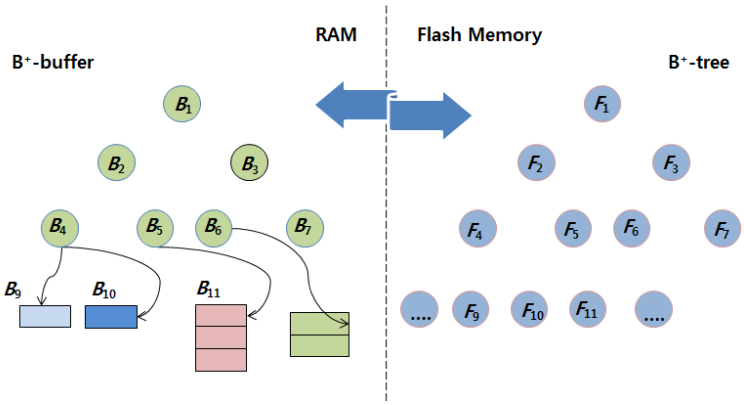


Fig. 2: Diagram of the proposed design.

Table 2: Parameters used in this paper

Parameter	Description
S	the total number of keys
N_i	a logical node of a B$^+$-tree node
F_i	a physical node of the B$^+$-tree, it can be referred to as the base node
B_i	a buffer unit of a corresponding B$^+$-tree node
B_i^c	a buffer unit of clean state
B_i^{sc}	a buffer unit of semi-clean state
C_i	an in-memory copy of the base node F_i
D_i	the changes made to the base node F_i
B_i^d	a buffer unit of dirty state
w	the window size that determines the ratio of the cold region to the total buffer
max#ofKey	the maximum number of keys that a B$^+$-tree node can have
#ofNentries	the information about how many index entries are stored in a B$^+$-tree node
#ofBentries	the information about how many index entries are stored in a B$^+$-buffer unit
#ofNWrites	the number of nodes that will be overwritten when B_i is flushed

Like other flash-aware index solutions, our design also stores the main index structure in flash-based storage and adopts an in-memory buffer designed to cope with the flash constraints and reduce the latency and energy consumption. However, the proposed design differs in one essential way: it has a B$^+$-tree structured buffer, called *B$^+$-buffer*. This has a twofold significance. First, since B$^+$-tree provides a data management method, key entries can be efficiently maintained as the buffer size increases, which is both convenient and efficient. Second, because keys are kept in a sorted order, we can perform a partial flush, thus affecting only a certain part of the B$^+$-tree index.

In the buffer, we generally attach one *buffer unit*, B_i, to increase the overall hit ratio and retain any changes made to B_i. At the proper time, all changes will be flushed out in

a batch to the *base node*, F_i, stored in flash memory. In this way, many small modifications are translated into one relatively larger write, which definitely achieves a better amortized write cost. We now formally define the following:

Definition 1 (Base node). *A base node, F_i, is a B+-tree index node stored in flash memory. It is made up of a set of a pointer, p_0^f, and n base entries. A base entry, denoted by x_j^f, consists of a search-key value, k_j^f, and a pointer, p_j^f, to a record with that value as its search-key value. Written more formally,*

$$F_i = \{p_0^f, x_1^f, x_2^f, \dots\dots, x_{n-1}^f, x_n^f\}, \tag{3.1}$$

where $x_j^f = \{k_j^f, p_j^f \mid j = 1, 2, \dots\dots, n\}$.

Definition 2 (buffer unit). *A buffer node, B_i, residing in the main memory is defined by two components: a clean part and a dirty part. The clean part is a copy of the corresponding base node, denoted by C_i, whereas, the dirty part changes to C_i, denoted by D_i. A change can be an insertion of a new base entry, a deletion of an existing base entry, or a modification of p_0^f or an existing base entry. A change is reflected as an index unit ${}_dx_j^b$, which consists of an operation-type indicator o_j, a search-key value ${}_dk_j^b$, and a pointer ${}_dp_j^b$ to a record. Formally, we define C_i, D_i, and B_i.*

First,

$$C_i = \{ {}_cp_0^b, {}_cx_1^b, {}_cx_2^b, \dots\dots, {}_cx_{n-1}^b, {}_cx_n^b \}, \tag{3.2}$$

where ${}_cp_0^b = p_0^f$ and ${}_cx_j^b = x_j^f$, for $j = 1, \dots, n$.

Second,

$$D_i = \{ {}_dp_0^b \} \cup \{ {}_dx_j^b \}, \tag{3.3}$$

where ${}_dp_0^b \neq p_0^f$ and

$${}_dx_j^b = \{o_j, {}_dk_j^b, {}_dp_j^b \mid o_j \in \{insert, delete, modify\}, j \in N, \forall j \forall k[(o_j = insert) \Rightarrow ({}_dk_j^b \neq k_k^f)], \forall j \exists k[(o_j = delete \vee modify) \Rightarrow ({}_dk_j^b = k_k^f)]\}.$$

Finally,

$$B_i = \{C_i\} \cup \{D_i\}. \tag{3.4}$$

Definition 3 (Logical node). *A logical node, N_i, is an up-to-date version of the base node F_i by applying changes. It is formally defined by*

$$N_i = \{x_k^f \in F_i\} + \{ {}_dx_j^b \in D_i \mid o_j = insert \vee modify\} - \{x_k^f \in F_i \mid \exists j(k_k^f =$$

$${}_{d}k_j^b, o_j = delete \vee modify)\}. \tag{3.5}$$

Lemma 1 (Exclusive Disjunction Output). *The logical node, N_i, is the result of the exclusive disjunction (XOR) of the base node, F_i, and the dirty part of the corresponding buffer unit:*

$$N_i = F_i \oplus D_i. \tag{3.6}$$

Proof. Assume that $x_k^f \in F_i$ and ${}_{d}x_j^b \in F_i$. Considering the previous three definitions, we can draw an output/truth table for them as follows:

Table 3: Index entry output table

Operation	Input		Output
	base entry	index unit	index entry
N/A	0	0	0
N/A	x_k^f	0	x_k^f
Insert	0	${}_{d}x_j^b$	${}_{d}x_j^b$
Delete	x_k^f	${}_{d}x_j^b$	0
Modify(Delete)	x'^f_k	${}_{d}x'^b_j$	0
Modify(Insert)	0	${}_{d}x''^b_j$	${}_{d}x''^b_j$

Note that a modification can be considered as a transaction consisting of the deletion of an old entry (${}_{d}x'^b_j$ in Table 3), and an insertion of a new entry (${}_{d}x''^b_j$ in Table 3). As we can see, two inputs produce a value of true (a non zero indicates that an output exists) if one, but not both, of them is true. The lemma immediately follows.

3.1.2 Three-State Scheme

Unlike methods proposed in previous studies, our approach does not simply classify the cached nodes into clean or dirty nodes. In the B$^+$-buffer, buffer unit B_i can have three states. We can define these states as follows:

Definition 4 (Clean State). *The clean unit, denoted by B_i^c, is a buffer unit that does not contain any changes to the corresponding base node, F_i. Formally,*

$$B_i^c = C_i. \tag{3.7}$$

Definition 5 (Semi-clean State). *If B_i consists of a copy of the corresponding base node, F_i, and its changes, it is said to be a semi-clean unit, denoted as B_i^{sc}. Formally,*

$$B_i^{sc} = C_i \cup D_i. \tag{3.8}$$

Definition 6 (Dirty State). *If B_i contains only the changes to the corresponding base node, F_i, it is said to be a dirty unit, denoted as B_i^d. Formally,*

$$B_i^d = D_i. \tag{3.9}$$

Using Definitions 2, 4, 5, and 6, and Lemma 1, we can easily deduce the following corollary.

Corollary 1.

$$N_i = \begin{cases} B_i & if\ B_i = B_i^c \\ B_i\ or\ C_i \oplus D_i & if\ B_i = B_i^{sc}. \\ B_i \oplus F_i & if\ B_i = B_i^d \end{cases} \tag{3.10}$$

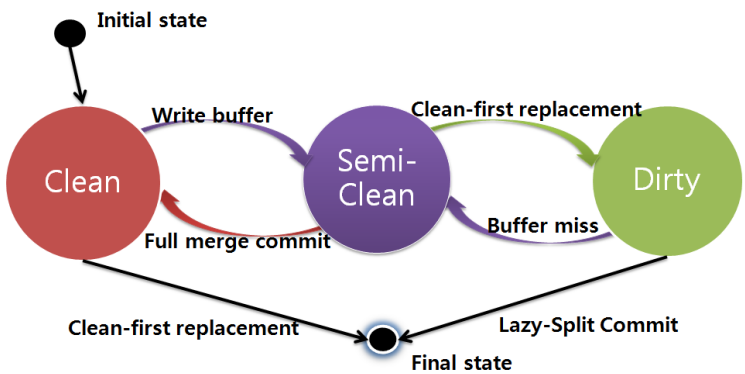


Fig. 3: A state diagram for a B+-buffer unit.

Figure 3 describes the lifetime of buffer unit B_i from the creation to eviction. The initial state is clean state B_i^c. After the write buffer stage, B_i^c will transition into semi-clean state B_i^{sc}. Since we use the *clean-first* strategy, B_i^{sc} can never return to clean state B_i^c unless we synchronize the buffer unit B_i and the base node F_i for the sake of power recovery. This is denoted by the arrow between the two states labeled *full merge commit*. The existing clean-first policy deals only with the victims of the clean state. However, we improved it further: the policy also transforms semi-clean state B_i^{sc} into dirty state B_i^d by flushing out the clean part of B_i^{sc}, which is termed *clean-part only commit*. A buffer miss brings a certain base node F_i into the buffer. If there already exists a corresponding dirty part, B_i^d, it converts into a semi-clean state B_i^{sc}; otherwise, the creation of a buffer unit, B_i,will take place.

3.2 A Lazy-Split Replacement Policy

As mentioned earlier, we also seek to address the asymmetric read/write speed of

flash by allowing dirty parts to stay longer in the B^+-buffer than clean parts. As can be seen from Fig. 4, the LS-LRU list is divided into hot and cold regions. In the *hot region*, most of the buffer hits occur, whereas in the *cold region*, the clean parts are always selected as victims over dirty parts. We keep two lists for the buffer units in the cold region: if a buffer unit is a clean unit or a semi-clean unit, it goes to the *clean list*; otherwise, it goes to the *dirty list*. Upon a buffer miss, if the clean list is not empty, the tail of the clean list is returned as the victim. Otherwise, the dirty units will be evicted by our *lazy-split commit* policy.

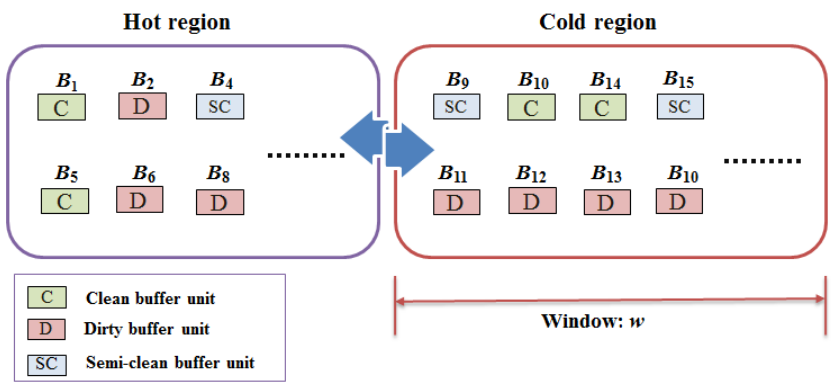


Fig. 4: LS-LRU buffer management scheme.

The LS-LRU behaves in the same manner as the previous methods in terms of the hit ratio and flash write count, but we set varying priorities for the buffer units. Specifically, we assign a higher priority to clean and semi-clean units over dirty units. In the case of an eviction of dirty units, we use two measurements (how many entries are contained and how many writes that can occur) to select the optimal victim. In contrast, the state-of-the-art methods [5, 7, 8] do not consider the discrepancy in the costs of the dirty units, they just evict the dirty unit at the tail of the LRU dirty list. Hence, our solution can simply make further improvements to the replacement cost of the dirty units.

When replacing dirty units, an important technical issue is how to efficiently merge cached index entries into a few base nodes of B^+-tree. That is to the manner of keeping the number of writes to flash at a minimum while flushing as many index entries as possible.

There are many index buffer replacement policies, such as *FIFO* (First In, First Out), a *First-fit commit* policy [8], a *First-associated commit* policy [15], and an *Exact-fit commit* policy [14]. They all explore a method for a minimization of the flash I/O under circumstances in which a B^+-tree split is performed normally.

Our previous study [16] suggested *modify-two-node* and *lazy-coalesce* policies to avoid the high cost of splitting and merging operations of B^+-tree. In this section, we further investigate a *lazy-split commit* policy that intelligently flushes those dirty nodes that might statistically cause fewer split operations. In the following, we first give a full description of the replacement algorithm, and then elaborate on two cases of commits.

Algorithm 1. The Lazy-Split Commit Policy

Input: a parameter μ, denotes a victim unit
Output: none

```
1:   if μ is null then          /* it means that the B+-buffer is full*/
2:       let θ denote the set of the disjoint sets of buffer units
3:       Create Statistical Table
4:       Use dynamic programming to choose a victim set from θ, denoted as α
5:       Let v be the set of the corresponding nodes of α
6:       MergeSort (α, v)
7:   else                        /* it means that a certain buffer unit is full */
8:       Let v be the corresponding node of μ
9:       MergeSort (μ, v)
10:  Flush ()
11:  end if
```

3.2.1 Write-Minimized Commit

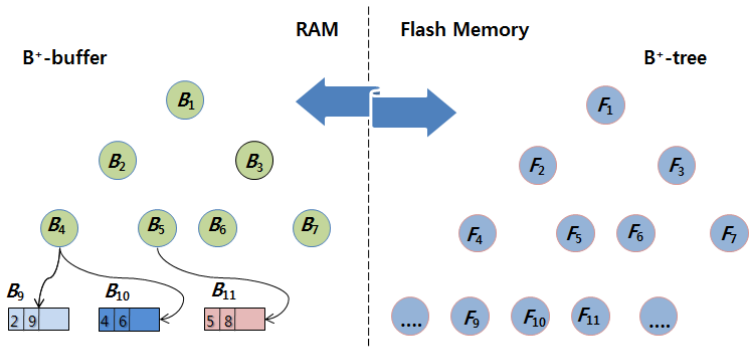
If the whole B+-buffer is above a threshold, some victims are selected for vacating the memory space. This is very common when keys are randomly distributed. We call this *write-minimized* commit, and it is processed as shown in lines 1 through 6 of Algorithm 1. If there is no designated victim, the buffer manager creates a *statistical table* and then selects a victim set dynamically. In the following, we first describe the key insight behind this, and then show the time complexity and how to construct an optimal solution.

Key Insight:

As shown in Fig. 5, each buffer unit for a B+-tree node, denoted by B_i, keeps information regarding the number of index entries stored in base node, F_i, and the number of index entries stored in the corresponding buffer node, B_i, which are represented by *#ofNEntries* and *#ofBEntries*, respectively. For instance, B_9 must have two entries, and F_9 must have nine, according to the information of B_9. When the B+-buffer is full, the commit policy will create a statistical table with the two fields, *#ofBEntries* and *#ofNWrites*. Here, *#ofBEntries* stores the number of index entries in B_i, and *#ofNWrites* stores the number of nodes that will be overwritten when B_i is flushed. The first field value is directly copied from B_i, and the second field can be derived from the following,

$$(\#ofBEntries + \#ofNEntries)/max\#ofKey + 2. \tag{3.11}$$

For example, B_{10} has *info (4, 1)*, which means that flushing B_{10} will issue one write to flash memory, and *info (5, 3)* of B_{11}, which means that flushing B_{11} overflows F_{11}, causing three writes to flash memory.



Buffer unit	Info (#ofBEntries, #ofNWrites)
B_9	(2, 3)
B_{10}	(4, 1)
B_{11}	(5, 3)
B_{12}	(3, 3)
B_{13}	(2, 1)

Statistical Table

Fig. 5: An example of five buffer units, and the commit policy for flushing B_{10} and B_{13}.

Originality:

We compare our commit policy with previous flash-aware index buffer schemes that do not consider the splitting problem. However, most of them store changes in a delta page in flash memory. They usually adopt the *first-fit commit* policy, as they do not really perform splitting. They avoid splits by writing more logs on the changes. However, the number of writes can be little reduced as the splits are delayed rather than eliminated. In addition, the search time and main memory usage will greatly increase since they will visit all delta pages (the common value of *max#ofKey* is 63 in many designs) where the related index entries reside to rebuild a logical B^+-tree node. The other approaches either simply use the *first-associated commit* policy, which chooses the first index entry in the buffer and collects its related index entries as victims [15], or are only concerned with space utilization [6, 14]. For example, they may commit B_9 to a flash device; as a result, three writes are required. That is, the replacement decision is not optimal; it would be better to commit B_{10}, which incurs one write to flash memory while flushing four key entries. To the best of our knowledge, LS-LRU is the first replacement algorithm that considers the costly splits of a flash-based B^+-tree index.

Time Complexity:

Our *write-minimized commit* policy can be defined as a *victim-choose* problem: a given set of buffer units, each with a weight and cost, determines the count of each buffer unit to include in a collection such that the total weight is at least equal to the given limit L, and the total cost C is minimal. Obviously victim-choose problem is NP-Complete.

Optimization Problem:

We have many approximation algorithms for solving the victim-choose problem, and there is a pseudo-polynomial time algorithm that uses *dynamic programming* [17]. If the subsets are optimal, the combined subset will be optimal. Hence, we can use the recursive property of dynamic programming to construct optimal solutions for increasingly larger sets in a bottom-up fashion. From this, each solution obtained will always be optimal.

3.2.2 Multi-Split Commit

The second case is committing when a certain buffer unit is full (lines 7 through 10). Based on our *lazy-split* policy, we split the nodes as lazily as possible. To this end, we allow the size of the buffer to grow to k ($k > 0$) times of that of the corresponding node (a B^+-buffer size is about one-tenth of B^+-tree in our implementation). The commit policy will flush the index entries of this buffer unit and empty it. This flush may cause the corresponding node to overflow, and it therefore needs more than two nodes to accommodate all key entries. We call this *multi-split commit*. The number of required leaf nodes, m, is derived from the following equation:

$$\lceil k \rceil - 1 \leq m \leq \lceil k \rceil + 1. \tag{3.12}$$

We illustrate the benefits of *multi-split commit* in Fig. 6. We assume that *max#ofKey* is 10 and k is 3, and thus B_{11} can hold at most 30 index entries. If we immediately conduct the splitting in flash, as a worst case, three nodes have to be written for every five incoming keys. In our example, the immediate split approach will cause 18 writes. In contrast, *multi-split commit* does not flush the index entries until B_{11} is full. Upon deciding to commit the buffer, LS-LRU will merge B_{11} with F_{11}, and write at most $\lceil 3 \rceil + 1 = 4$ leaf nodes and one internal node. This can save thirteen writes.

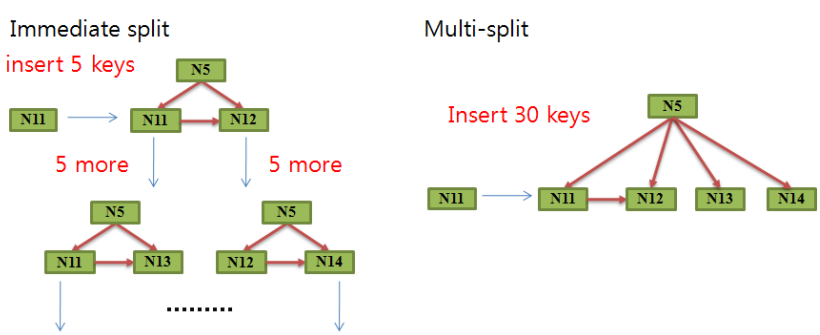


Fig. 6: An example of *multi-split commit*.

3.3 Full Merge Commit

For an index to be able to recover from a crash, we need to restrict the time when a buffer unit may be written back to flash. In addition, the *clean-first* strategy raises a

problem of information redundancy, as the buffer index entries of a delete type cannot be neutralized with the corresponding keys that reside in flash memory. To overcome these problems, LS-LRU is designed to commit the entire B^+-buffer periodically and manually. To do so, we use *Mergesort* [17], which combines two sorted inputs, a B^+-buffer unit, B_i, and a B^+-tree node, F_i, into one sorted output. Mergesort involves three steps: (1) dividing the keys into two parts, each with one-half of the key entries; (2) using a recursion to conquer each part by sorting it, unless the result is sufficiently small; and (3) combining the results by merging them into a single sorted-key array. When applying Mergesort to LS-LRU, the first two steps are already achieved: the keys are divided into leaf buffer units and nodes, and every buffer unit or index node is trivially sorted out. Our solution is therefore amenable to immediately start the merging part. By repeatedly calling the merging procedure, we can obtain the sorted key entries.

Algorithm 2 illustrates the *full merge commit* procedure.

Algorithm 2. Full Merge Commit Policy

Inputs: κ, the first leaf unit of B^+-buffer
χ, be the first leaf node of B^+-tree
Outputs none

```
if (κ != null and χ != null) then
        MergeSort (κ, χ)
        κ = κ.next
        χ = χ.next
Flush ()
end if
```

This illustrates a simple and effective method of intersecting key entries: we walk through the two key entry lists simultaneously in time linear to the total number of key entries. We compare the key values at each step. If they are not the same, we put the smaller key into the results list, and advance the corresponding pointer. Otherwise, we ignore them and advance both pointers. If the lengths of the lists are i and j, the intersection takes $O(i + j)$ operations. Formally, the time complexity is $O(S)$, where S is the total number of key entries.

Time Complexity:

Let us analyze the worst-case time complexity of a full commit policy. If the lengths of the lists are i and j, the intersection takes $O(i+j-1)$ operations, and they are recursively called p times, where p is the number of leaf nodes of the B^+-tree. The total time complexity of the full commit policy is therefore

$$T(\textit{full merge}) = O(i + j - 1) \times O(p). \tag{3.13}$$

The result of the multiplication of the right side is around S, and thus we have

$$T(\textit{full merge}) \leqslant O(S). \tag{3.14}$$

We conclude that the running time of a full commit policy is $O(S)$. This is proof of the advantage of maintaining an in-memory buffer in a B^+-tree fashion.

4. PERFORMANCE EVALUATION

In this section, we summarize the performance simulations. We first present our test environment and simulation methodology. We then compare our proposal with pure LRU and CFDC [6], measuring the performance of B^+-tree index operations under varying workloads.

4.1 Test Environment

The evaluation was performed on a Core i5-2500 machine (3.30GHz, 4GB DDR2 RAM) running the Ubuntu 10.10 operating system (Linux 2.6.35). We assumed that the embedded system has 128 MB of RAM and 16 GB of NAND flash memory, and we used the weighted costs of read, write, and erase operations (cf. Table 1). The reason why we chose CFDC as a major competitor is because it adopts the clean-first replacement policy, as in all other related works, and applies itself to a flash-based B^+-tree index, as in the present research. We integrated LS-LRU with pure LRU and CFDC into the buffer manager, and also implemented a B^+-tree index structure using 4 bytes of unsigned integers for the keys and pointers. The system was programmed and compiled using a GNU C++ compiler. All keys were chosen randomly with a uniform distribution. The node size was the same as the size of the flash page. At the beginning of each test, the indices were populated with 100,000 keys. We statically set the window size w to 0.5. We also measured the time that elapses from start to finish, including the time for programmed delays or from waiting for resources to be assigned, using a wall clock.

The evaluation was conducted under two types of workload. The first type contains three different synthesized traces used for a better control of the workload configuration, as shown in Table 4. The second type contains two realistic configurations, namely OLTP-A and OLAP-B which are B^+-tree index operation traces collected from the inside of a MySQL DBMS running an industry-academic cooperation system. Table 5 shows details on the real traces used.

Table 4: Three types of synthesized profiles used in the simulation

	Select	Insert	Delete	Modify	Total
Write-intensive	10,000	30,000	30,000	30,000	100,000
Read-intensive	90,000	3,000	3,000	4,000	100,000
Random	N/A	N/A	N/A	N/A	100,000

Table 5: Characteristics of the real traces

	# Reference	Write ratio	Description
OLTP-A	1,264,928	74.1%	transactional queries
OLAP-B	689,486	1.5%	analytical queries involving aggregations

All of these profiles mix the point queries and range queries, as shown in Table 6. A point query randomly manipulates a certain key value and its pointer, whereas a range query sequentially manipulates a certain number of keys. Note that a modify operation here simply finds a given key and updates its pointer.

Table 6: Specifications of the index operations

Point query	Search	Search for a certain key
	Insertion	Insert a certain key
	Deletion	Delete a certain key
	Modification	Update the pointer of a certain key
Range query	Search	Search for a range of keys
	Insertion	Insert a range of keys
	Deletion	Delete a range of keys
	Modification	Update the pointers of a range of keys

4.2 Experimental Results

4.2.1 Varying Read/Write Ratio

Most database systems experience a varying read-write ratio over time, but even when the workload of an application is write-only, the workload experienced by the index buffer cannot be. Since the manipulation of a B^+-tree requires a lookup operation to locate where to apply the change, and the index buffer is a part of the B^+-tree structure, we have to look into it first. We therefore vary our B^+-buffer experienced ratio, as shown in Fig. 7, where the X-axis stands for the read-write ratio, and the Y-axis indicates the accumulated time in microseconds. This figure shows the effect of increasing the write operation ratio. Under a heavy-read workload (e.g., 100/0 and 90/10), CFDC and LS-LRU degenerate into LRU. However, it is clear that the pure LRU is not optimal with a flash write workload. Compared to LRU, CFDC is expected to perform fewer page flushes, which explains its slight performance improvement over LRU. The LS-LRU clearly outperforms the others, mostly because it generates fewer splits, and benefits from the efficient three-state scheme of the buffer units, which can limit obsessive lookups and updates in the buffer. As the best case, the access performance is improved by 40.4% and 25.4%, compared to LRU and CFDC, respectively, under a read-write ratio of 50/50.

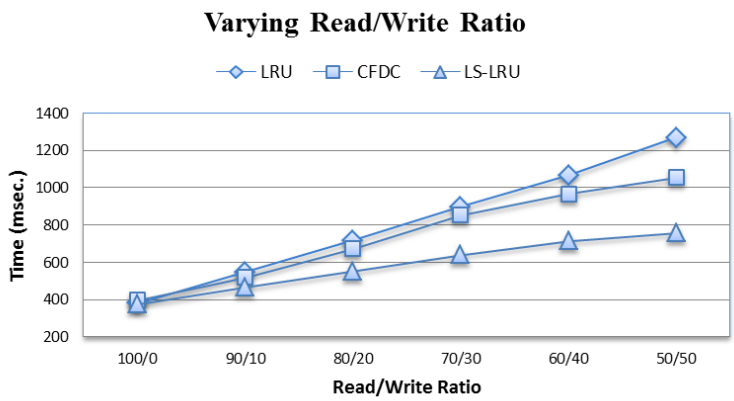
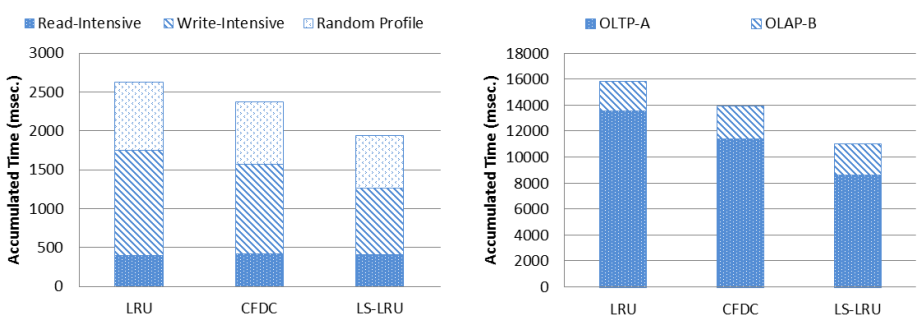


Fig. 7: Effects of increasing a random write operation ratio.

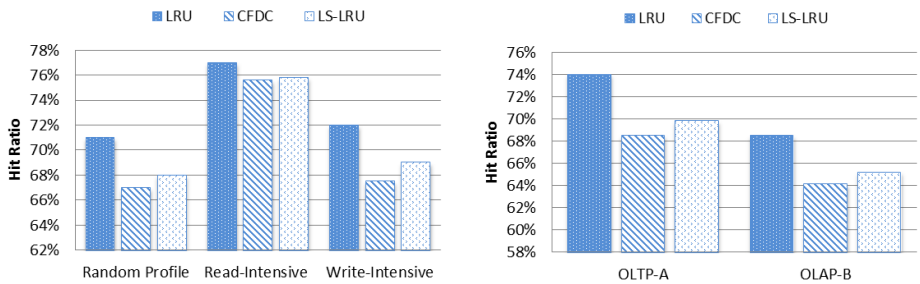
4.2.2 Comparison of Execution Times

Fig. 8 compares the total execution times of three algorithms under both synthesized and real traces. The performance is estimated as the sum of flash memory access time and main memory reference time. LS-LRU provides the best overall performance gain. Specifically, compared to LRU and CFDC, the cost of LS-LRU is reduced by 38.8% and 25.0% under a write-intensive workload, and by 36.3% and 24.3% under an OLTP-A workload, respectively. Both flash writes and runtime were greatly reduced by LS-LRU. In the case of a random workload, LS-LRU outperforms both competing policies, with a performance gain of 17.0% over CFDC, which is only slightly better than LRU with a performance gain of 6.3%. Again, there are hardly any observable differences among them under read-intensive OLAP-B workloads.



(a) synthesized traces (b) real traces

Fig. 8: Overall runtime of LRU, CFDC, and LS-LRU under both synthesized and real traces.



(a) synthesized traces (b) real traces

Fig. 9: Buffer hit ratios of LRU, CFDC, and LS-LRU under both synthesized and real traces.

4.2.3 Buffer Hit Ratio

We counted the buffer references that were measured through the previous tests. These measurements are shown in Fig. 9; in terms of the buffer hit ratio, CFDC and LS-LRU have small performance degradations when compared with the pure LRU, which are due to more buffer misses that can occur when we adopt a clean-first strategy. Nevertheless, LS-LRU still has the best performance under balanced and heavy-write workloads, as it minimizes the number of writes to flash memory.

4.2.4 Varying Window Size

Finally, Fig. 10 shows the effects of LS-LRU and CFDC with varying sizes of window w under two real traces. When w is 0, LS-LRU and CFDC perform the same as LRU, but when w is 1, LS-LRU and CFDC consistently perform a clean-first eviction. The execution times shown on the Y-axis are normalized using the LRU results. As shown in OLTP-A trace, the performance gains of CFDC and LS-LRU are quite significant. The larger the size of the cold region, the fewer pages that CFDC and LS-LRU flush. In the case of a heavy-read workload of OLAP-B trace, LS-LRU and CFDC no longer outperform LRU when w is above 0.75, as the clean-first eviction harms the buffer hit ratio when w is large. However, the performance gaps among them are not big, since most of the cached pages are clean. LS-LRU shows a better performance than the others, and most of the gains come from a better amortized write cost owing to he transformation of many small modifications into one relatively large write. Roughly speaking, compared to LRU and CFDC, the cost of LS-LRU is at best reduced by 43.9% and 23.8% for OLTP-A with window size w = 1, and by 6.3% and 1.5% for OLAP-B with w = 0.5, respectively.

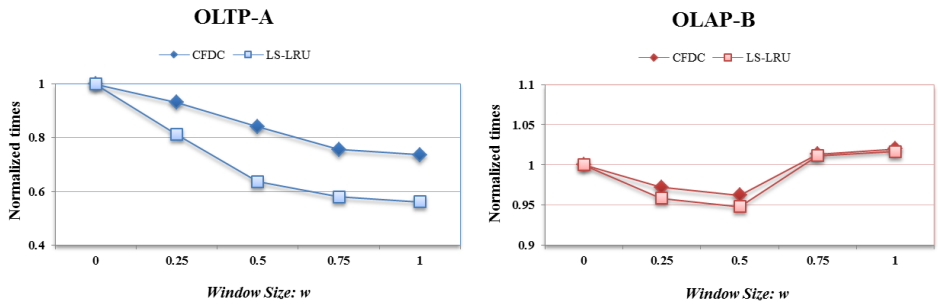


Fig. 10: Comparison of normalized execution times in CFDC and LS-LRU with varying window size *w*, under two real traces.

5. CONCLUSIONS AND FUTURE WORK

Despite its many advantages, the existing algorithms perform poorly with flash memory, which also has many unique electronic limitations. In consideration of the relative costly write speed of flash memory, researchers have proposed flash-aware replacement algorithms during the past few years, based on a simple idea of reducing the number of write operations by first replacing clean pages. However, each of these has failed to find an effective solution to reduce intensive overwrites that are caused by a frequently changing index structure. As an improvement to this, we proposed Lazy-Split LRU (LS-LRU), a three-state buffer management scheme to improve the buffer space utilization and further reduce flash writes compared with other related methods. Moreover, we introduced a family of efficient commit policies that can reduce the number of B^+-tree splits.

A trace-driven evaluation showed that LS-LRU significantly outperforms pure LRU and CFDC under balanced and heavy-write workloads. In detail, LS-LRU improves the access performance of flash-based B^+-tree by 20 to 40% under a heavy-write trace, and by 1 to 6% under a heavy-read trace. In addition, our solution is expected to obtain a good reclamation performance because it limits most obsessive modify operations in the buffer. This kind of measurement will be a part of our future work. Additionally, to further improve the performance of B^+-tree in flash memory, we are going to explore new storage and index managers that will collaborate closely with the purposed buffer management scheme in flash memory storage systems.

REFERENCES

1. Samsung Electronics, NAND based solid state drive, *SSD (MMDPE56G8DXP-0VB) data sheet*, 2012.
2. T. S. Chung, D. J. Park, S. W. Park, D. H. Lee, S. W. Lee, and H. J. Song, "A survey of Flash Translation Layer," *Journal of Systems Architecture*, vol. 55, no. 5-6, 2009, pp. 332-343.

3. S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. W. Park, and H. J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. on Embedded Comp. Syst.*, vol. 6, 2007.
4. S. W. Lee and B. Moon, "Design of flash-based dbms: An in-page logging approach," in *Proceedings of the ACM SIGMOD*, 2007, pp. 55–66.
5. S. Park, D. Jung, J. Kang, J. Kim, and J. Lee,, "CFLRU: a replacement algorithm for flash memory," in *Proc. Of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Seoul, Korea October 2006, pp. 234–241.
6. Y. Ou, T. Härder, and P. Jin, "CFDC: a flash-aware replacement policy for database buffer management," in *DaMoN*, 2009, pp. 15–20.
7. X. Tang, X. Meng, "ACR: an Adaptive Cost-Aware Buffer Replacement Algorithm for Flash Storage Devices," in *Proceedings of the International Conference on Mobile Data Management (MDM '10)*, Kansas City, Missouri, USA, 2010.
8. C. H. Wu, T. W. Kuo, and L. P. Chang, "An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems," *ACM Trans. on Embedded Comp. Syst.*, vol. 6, no. 3, July 2007, pp. 1-23.
9. J. Park, E. Lee, and H. Bahn, "DABC-NV: a buffer cache architecture for mobile systems with heterogeneous flash memories," *IEEE Trans. Consumer Electron.*, vol. 58, no.4, pp. 1237-1245, Nov. 2012.
10. B. Kim and D. Lee, "LSF: a new buffer replacement scheme for flash memory-based portable media players" *IEEE Trans. Consumer Electron.*, vol. 59, no.1, pp. 130-135, Feb. 2013.
11. D. Seo and D. Shin, "Recently-evicted-first buffer replacement policy for flash storage devices," *IEEE Trans. Consumer Electron.*, vol. 54, no. 3, pp. 1228–1235, Aug. 2008.
12. X. Liao and S. Hu, "Bridging the information gap between buffer and flash translation layer for flash memory," *IEEE Trans. Consumer Electron.*, vol. 57, no. 4, pp. 1765-1773, Nov. 2011.
13. S. Nath and P. B. Gibbons, "Online maintenance of very large random samples on flash storage," *VLDB Journal*, vol. 19, no. 1, 2010, pp. 67-90.
14. M. V. Jørgensen, R. B. Rasmussen, S. Šaltenis, and C. Schjønning, "FB-Tree: A B^+-Tree for Flash-Based SSDs," in *Proceedings of the 15th Symposium on International Database Engineering & Applications (IDEAS '11)*, Lisbon, Portugal, 2011, pp. 34-42.
15. H. S. Lee, and D. H. Lee, "An efficient index buffer management scheme for implementing a B-tree on NAND flash memory," *Data Knowl. Eng.*, vol. 69, 2010, pp. 901-916.
16. R. Jin, S. J. Kwon, and T. S. Chung, "FlashB-tree: a novel B-tree index scheme for solid state drives," in *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, Miami, FL, USA, November, 2011, pp. 50-55.
17. R. Neapolitan and K. Naimipour, *Foundations of Algorithms*. 4th ed., Jones & Bartlett Publishers, USA. Dec. 31st, 2009.

BIOGRAPHIES

Rize Jin received his B.S. degree in computer science and technology from Hunan University of Commerce, Changsha, China in 2008, and his M.S. degree in computer engineering from Ajou University, South Korea, in 2011. He is currently a Ph.D. candidate at the Dept. of Computer Engineering, Ajou University. His research interests include memory and storage algorithm design, flash-based DB systems, FTL, and location based services.

Hyung-Ju Cho received his B.S. and M.S. degrees in computer engineering from Seoul National University in February 1997 and February 1999, respectively, and his Ph.D. degree in computer science from KAIST in August 2005. He is currently a research assistant professor at the Dept. of Information and Computer Engineering, Ajou University, South Korea. His current interests include flash memory, large database systems, moving object databases, and query processing in mobile peer-to-peer networks.

Tae-Sun Chung received his B.S. degree in computer science from KAIST, in February 1995, and his M.S. and Ph.D. degrees in computer science from Seoul National University, in February 1997 and August 2002, respectively. He is currently a professor at the Dept. of Information and Computer Engineering, Ajou University. His current research interests include flash memory storage, XML databases, and database systems.