μ^*-Tree: An Ordered Index Structure for NAND Flash Memory with Adaptive Page Layout Scheme

Jung-Sang Ahn, Dongwon Kang, Dawoon Jung, Jin-Soo Kim, *Member*, *IEEE*, and Seungryoul Maeng

Abstract—As NAND flash memory is gaining popularity as a storage medium for mobile embedded devices, many flash-aware file systems, flash-aware DBMSes, and flash translation layers (FTLs) require an flash-efficient index structure. This paper proposes a novel index structure called μ^*-Tree which natively works on NAND flash memory, aiming at improving performance over B^+-Tree. μ^*-Tree stores all the nodes along the path from the root to the leaf into a single flash memory page in order to minimize the number of flash write operation when a node is updated. Furthermore, μ^*-Tree has an adaptive page layout scheme which dynamically adjusts the page layout according to the workload characteristics on-the-fly. μ^*-Tree also allows flash pages with different page layouts to coexist in the same tree. Our evaluation results with real workload traces show that μ^*-Tree outperforms B^+-Tree by up to 55 percent in terms of the time needed for flash operations. With a small in-memory cache of 32 KB, μ^*-Tree improves the overall performance by up to five times compared to B^+-Tree with the same cache size.

Index Terms—NAND flash memory, index structure, B^+-Tree

1 Introduction

NAND flash memory is being widely used as a storage medium in modern mobile embedded devices such as MP3 players, PMPs, and cell phones. Recently, NAND flash-based Solid State Drives (SSDs) are rapidly becoming viable competitors to Hard Disk Drives (HDDs) in notebook and desktop PC markets due to their advantages in terms of performance, energy consumption, weight, shock resistance, etc.

However, NAND flash memory features several unique characteristics. First, once data are written in NAND flash memory, it cannot be overwritten until it is erased. Second, the unit of erase operation (called *block*) is larger than the unit of read and write operations (called *page*) by $64 \sim 256$ times. Third, the number of erase operations that can be performed on a single block is limited.

In legacy systems, the unique characteristics of NAND flash memory are hidden via a software layer called FTL (Flash Translation Layer) [1], [2], [3]. One of the main functionalities of FTL is *address translation*, the ability to write incoming data into any preerased location by keeping track of mapping information between logical pages and the corresponding physical pages in NAND flash memory. In this way, FTL provides the general block device interface, on top of which the existing disk-based file systems or DBMSes can be built. For flash memory cards (e.g., SD, e-MMC, etc.) and SSDs, FTL is located inside the device as a form of firmware. When NAND flash chips are directly attached to the system as in digital TVs and iPhones, FTL is implemented in the device driver of the host system.

Another approach to managing NAND flash memory is to use flash-aware file systems such as UBIFS, JFFS3, and YAFFS, or flash-aware DBMSes such as FlashDB [4], and LGeDBMS [5]. They understand the characteristics of flash memory, thus it is possible to work directly on flash memory without the need for FTL.

We note that all of the aforementioned approaches require an efficient index structure. Basically, an index is a data structure that enables sublinear-time lookup. Index structures are heavily used in file systems to map a pathname to a file metadata (i.e., inode) or to locate the data block associated with the given offset of a file. In DBMSes, various indexes are used to improve the speed of data retrieval operations on a database table. Moreover, extent-based FTLs such as μ-FTL [1] also employ an index structure for managing their address mapping information.

Indexes can be implemented using a variety of data structures, but B^+-Tree is one of the most popular index structures [6], [7]. B^+-Tree guarantees logarithmic access time and supports efficient retrieval in block-oriented storage by reducing the number of I/O operations through very high fanout. Although B^+-Tree is a successful index structure for disk-based storage, it is not the case for NAND flash-based storage; since in-place update is not allowed in flash memory, any change in a leaf node will propagate to the root node causing a lot of page writes.

- *J.-S. Ahn and S. Maeng are with the Department of Computer Science, KAIST, Daejeon 305-701, Korea. E-mail: {jsahn, maeng}@camars.kaist.ac.kr.*
- *D. Kang is with Google Korea LLC., Seoul, Korea. E-mail: dwkang@google.com.*
- *D. Jung is with the Memory Division, Samsung Electronics Corporation, Hwasung, Korea. E-mail: dw0904.jung@samsung.com.*
- *J.-S. Kim is with the School of Information & Communication Engineering, Sungkyunkwan University, Suwon, Korea. E-mail: jinsookim@skku.edu.*

Manuscript received 10 May 2011; revised 25 Dec. 2011; accepted 2 Jan. 2012; published online 16 Jan. 2012.
Recommended for acceptance by C.-W. Wu.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-05-0313.
Digital Object Identifier no. 10.1109/TC.2012.20.

To remedy this *wandering tree problem*, this paper suggests μ^*-Tree, a new ordered index structure tailored to the characteristics of NAND flash memory. μ^*-Tree is a variation of B^+-Tree, where all updated nodes from the leaf to the root are stored into a single page. In most cases, as only one page is written for each update in a leaf node, the overall update cost is improved compared to B^+-Tree. However, storing all updated nodes into a single page makes μ^*-Tree occupy more flash pages due to smaller leaf node size compared to B^+-Tree which uses the entire page as a leaf node. To overcome this space inefficiency, a set of additional schemes is proposed in this paper. First, μ^*-Tree generalizes the page layout so that it can store records in a fine-grained manner. Second, μ^*-Tree has an adaptive page layout scheme where different page layouts can coexist in the same tree. In particular, μ^*-Tree adjusts the page layout automatically according to the state of the tree on-the-fly so that we do not have to preset the page layout for dynamically changing workloads. Our experimental results with real workload traces show that μ^*-Tree outperforms B^+-Tree by up to 55 percent in terms of the time spent for flash operations. With a small in-memory cache of 32 KB, μ^*-Tree improves the overall performance by up to five times compared to B^+-Tree with the same cache size.

The rest of the paper is organized as follows: Section 2 overviews NAND flash memory and B^+-Tree, and presents the motivation of μ^*-Tree. Section 3 overviews μ-Tree, the previous version of μ^*-Tree. Section 4 describes the structure of μ^*-Tree and the adaptive page layout scheme. Section 5 presents the evaluation results, and Section 6 gives the related work. Section 7 concludes the paper.

2 Background and Motivation

2.1 NAND Flash Memory

NAND flash memory is a nonvolatile semiconductor device. The initial state of each NAND flash memory cell is set to "1," and a set of selected cells can be transitioned to "0" by write (or program) operation. Once the cell is programmed to "0," reverting to the original "1" state requires a special *erase* operation. This means that after we write certain data into NAND flash memory, we cannot overwrite another data in the same location until the original data are erased.

NAND flash memory has a hierarchical structure where a chip consists of a number of *blocks* and each block is, in turn, composed of $64 \sim 256$ *pages*. A page is basic unit of read and write operations, while erase operations are performed on a block basis. Thus, it is not possible to erase a particular page in a block selectively. In addition, NAND flash memory cells tend to wear out over time as they are repeatedly programmed and erased. Usually, the number of erase operations that can be performed on a single block is limited to $5{,}000 \sim 100{,}000$ times. This necessitates a technique called *wear-leveling* which avoids concentrating writes on particular blocks to extend the lifetime of flash memory.

NAND flash memory can be classified into two types: Single-Level Cell (SLC) and Multi-Level Cell (MLC) [8], [9]. In typical SLC NAND chips, the page size is 2 KB and a block consists of 64 pages. On the other hand, MLC NAND achieves higher density and larger capacity by storing two or more bits per each memory cell. As a result, the size of each page in MLC NAND is increased to $4 \sim 8$ KB, and the number of pages in a block is also increased to $128 \sim 256$ pages. Even though the operation latency of MLC NAND is longer than that of SLC NAND, many of NAND flash-based devices currently being used are based on MLC NAND due to its low cost-per-bit.

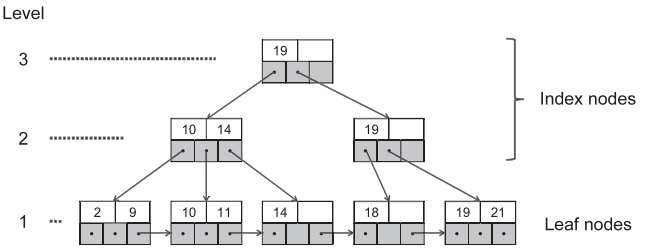


Fig. 1. An example of B^+-Tree of order 2.

2.2 B^+-Tree

B^+-Tree is one of the most popular index structures for managing a large amount of records. B^+-Tree is a balanced search tree which provides efficient insertion, deletion, and retrieval operations with the guarantee of amortized logarithmic access time. Many file systems (such as NTFS, ReiserFS, JFFS3, etc.) and DBMSes (such as MSSQL, MySQL, Oracle 8, etc.) are using B^+-Tree for metadata indexing or table indexing.

Unlike binary search tree, each node in B^+-Tree consists of a number of key and pointer pairs, while each node in a binary search tree has only one key and two pointers. A node in B^+-Tree contains up to d key values, $K_1, K_2, \ldots, K_d$, and $d+1$ pointers, $P_1, P_2, \ldots, P_{d+1}$. The key values in a node are kept in sorted order. Thus, if $i < j$, then $K_i < K_j$. The maximum number of keys that can be stored in a node, d, is called the *fanout* or *order* of B^+-Tree. Fig. 1 illustrates an example B^+-Tree of order 2. The high fanout degree of B^+-Tree has the advantage of reducing the number of I/O operations when accessing an entry in the tree. The size of each node is set to a multiple of sector size for HDDs or to a multiple page size for NAND flash memory.[1]

There are two types of nodes in B^+-Tree: leaf nodes and nonleaf nodes (or *index nodes*). Except for a tree consisting of a single node, a leaf node can store from $d/2$ to d keys and the associated pointers. For $i = 1, 2, \ldots, d$, each pointer P_i points to the record corresponding to the key K_i. P_{d+1} is usually used for chaining the leaf nodes in key order to facilitate range searches. For index nodes, the structure is essentially the same but pointers refer to other B^+-Tree nodes.

We define the *level* of a node as the number of edges on the shortest path from leaf nodes to the node plus one. According to this definition, the level of any leaf node is one, and the level of the root node is always the same as the height of the tree.

B^+-Tree tries to keep its nodes balanced whenever any insertion or deletion occurs. Unbalanced trees such as binary search tree may have a long path because they can be skewed by a certain insertion sequence. However, B^+-Tree guarantees the logarithmically bounded depth for all leaf nodes.

1. Without loss of generality, we assume that each node occupies a single flash page in this paper.

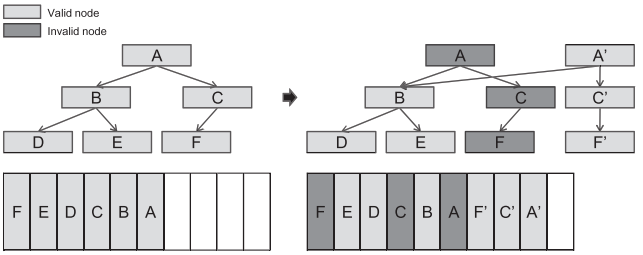


Fig. 2. An example of B^+-Tree update.

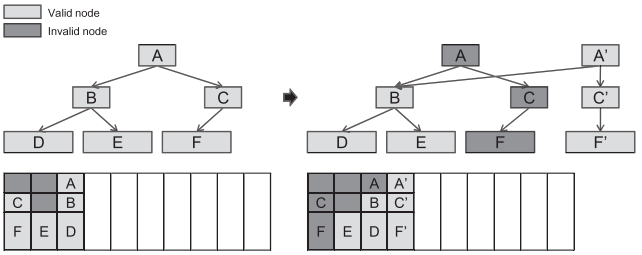


Fig. 3. An example of μ-Tree update.

To find a record with a key K in B^+-Tree, several nodes in a path from the root to the leaf are visited. At each node, the retrieval process searches for the smallest key K_i greater than K, and loads the node designated by the corresponding pointer P_i. If there is no such key, it follows the last pointer P_m, where m is the number of pointers in the node. When it reaches the leaf node, it checks whether the node contains the key K_i that is equal to K. If the node has such K_i, it returns the record pointed to by P_i. Otherwise, the retrieval process fails.

To insert a key K, the insertion process first finds the proper leaf node by using the above retrieval procedure. Then, it inserts the key K into the leaf node. If the leaf node is already full, a split occurs. In general, to handle $d+1$ keys, the first $\lceil (d+1)/2 \rceil$ keys are put into the existing node and the remaining keys into a new node. After the split, the lowest key of the new node is inserted to its parent node as a separator.

In most cases, the parent node is not full and the insertion process ends. If the parent node is full too, the split occurs again recursively up to the root node. When the recursive split process finally reaches the root node, B^+-Tree increases its height. This results in a new root node which has only one key and two pointers.

The deletion process for a key K proceeds similarly. After finding the proper leaf node, the key is removed from the leaf node. However, in order to keep the property that each node must have at least $d/2$ keys, balancing techniques such as redistribution or concatenation may occur [7]. As a result of concatenation, a node can be removed from B^+-Tree, and recursive concatenations may decrease the height of B^+-Tree by one.

For B^+-Tree of order d with n records, the cost of retrieval, insertion, or deletion operation is proportional to $\log_{d/2} n$ in the worst case. Thus, as the node size increases, the operation cost will drop due to the increased branching factor d. Actually, the optimal node size depends on the characteristics of the underlying system and storage devices. In many cases, B^+-Tree uses the node size ranging from 4 to 16 Kbytes for modern hard disks considering the seek penalty, the rotational delay, DMA granularity, the unit of paging, and many other factors [10].

2.3 Motivation

We encounter a problem when implementing B^+-Tree on NAND flash memory. Recall that all records inserted into B^+-Tree are pointed to by leaf nodes. To update a record, the corresponding leaf node should be written to another page because NAND flash does not support in-place update.[2] By writing the leaf node into a new page, its parent node also should be updated as one of its pointers has been modified. This recursive update ends when the new root node is written into another page.

Fig. 2 depicts an example of B^+-Tree whose height is three. If we want to update the leaf node F, the parent index nodes, C and A, also need to be updated, requiring the total three flash page writes. We can see that, to update a record in B^+-Tree, it is necessary to write as many flash pages as the height of the tree. This is very inefficient because write and erase operations are expensive in NAND flash memory.

The designers of JFFS3 have used an in-memory data structure called the *journal tree* to reduce the number of updates on index nodes. When something is changed in the JFFS3 file system, the corresponding leaf node is written into flash memory, but the index nodes are updated only in the journal tree. Periodically, those delayed updates are committed, i.e., written into flash memory in bulk. The journal tree allows to merge many index node updates and lessen the amount of flash write operations. However, the use of the journal tree is not appropriate for small embedded systems with limited memory size because it requires memory in proportion to the number of updates delayed. Even worse, the larger is the journal, the longer it may take to mount JFFS3 since the journal tree should be built from the uncommitted leaf nodes when JFFS3 is being mounted. This paper proposes a new novel index structure called μ^*-Tree, which requires only a single flash write operation whenever any node in the tree is updated.

3 μ-Tree

In our earlier work, we proposed μ-Tree [11] to implement an efficient index structure on NAND flash memory. Since the overall structure of μ^*-Tree is based on that of μ-Tree, we briefly describe the page layout of μ-Tree and its limitations in this section.

3.1 Page Layout

μ-Tree stores all the updated nodes along the path from the leaf to the root into a single flash memory page. Fig. 3 shows an example of μ-Tree, where the same leaf node F is updated as in Fig. 2. In μ-Tree, the modified parent nodes, C' and A' are also written in the same page together with F', thus guaranteeing only one flash page write for each update request.

While the node size is fixed and same for all nodes in B^+-Tree, the size of each node in μ-Tree varies depending on the level of the node and the current height of μ-Tree. When the height of μ-Tree is greater than one, each leaf node occupies half of a page. As the level of a node increases, the node size is reduced by half, and only the

2. Here, we assume that each node in B^+-tree occupies a single flash page.

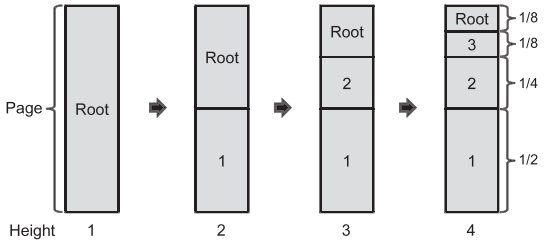


Fig. 4. The change in the page layout in μ-Tree.

root node has the same size as its child nodes. Suppose that the height of μ-Tree is H and the page size is P. The node size N_L at level L can be represented as follows:

$$N_L = \begin{cases} P & \text{if a single level } (H = 1); \\ P \cdot \left(\frac{1}{2}\right)^{L-1} & \text{if root } (L = H \text{ and } H > 1); \\ P \cdot \left(\frac{1}{2}\right)^{L} & \text{otherwise } (1 \leq L < H \text{ and } H > 1). \end{cases} \quad (1)$$

Fig. 4 illustrates the change in the page layout when we vary the height of μ-Tree from one to four.

The page layout used in μ-Tree ensures the same node size for each level regardless of the μ-Tree height. This is important especially when the height of μ-Tree grows or shrinks because all the existing nodes except for the root node can be reused. If the entire nodes were to be reorganized to fit into the new layout whenever the height is changed, it would not be acceptable due to considerable write operations.

3.2 Tree Operations

We describe the distinction between B^+-Tree and μ-Tree when they handle tree operations such as retrieval, insertion, and deletion. The major difference between B^+-Tree and μ-Tree is that the size of a node in μ-Tree is determined by its level and height.

Retrieval. The retrieval process of μ-Tree is basically same as that of B^+-Tree, because the logical structures of two trees are identical. The only difference lies in reading a node from pages. There are more than one nodes in a single page of μ-Tree, while there is only one node in B^+-Tree.

Insertion. The insertion process of μ-Tree first finds the target leaf node using the retrieval process. To insert a key into the leaf node, μ-Tree allocates a new page and the updated target leaf node is written into the page along with all the updated parent nodes. Unless there is an overflow in any of updated nodes, μ-Tree always writes only one page.

When one of updated nodes becomes full due to the insertion, μ-Tree splits the node. Since two different nodes at the same level cannot coexist in the same page, μ-Tree allocates one more page. Then, the former half of the node is written into the first page, and the latter half and all of its parent nodes into the second page. When the root node becomes full as a result of the insertion, the height of μ-Tree is raised by one.

Deletion. The deletion process of μ-Tree is similar to the insertion process. μ-Tree first finds the target leaf node, and deletes the key and the associated pointer from the node. All updated nodes from the leaf node to the root node are written into a newly allocated page. If the leaf node becomes empty by deletion, the leaf node is deleted and the pointer that points to the node in its parent node is also deleted. Deletion can be propagated to the root node, and the height of μ-Tree shrinks when the root node has only one pointer.

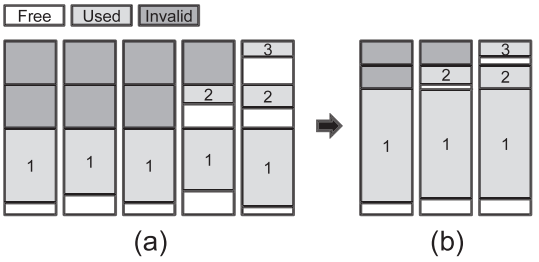


Fig. 5. μ-Tree examples: (a) the original layout and (b) the page layout with enlarged leaf nodes.

Note that leaf nodes are not chained together in μ-Tree. Chaining leaf nodes is useful for tree rebalancing and range lookups, but out-of-place update to a leaf node would lead to tremendous amount of consecutive updates through the links between leaves on NAND flash memory. In most cases, we can avoid additional flash overhead during range lookups using a small in-memory cache of recently-accessed pages [11], as the next leaf node can be retrieved through the parent node which is usually cached in memory.

3.3 Limitations and Opportunities

As mentioned in Section 3.1, only half of a single page is devoted to the space for storing pointers to records in μ-Tree. Consequently, μ-Tree requires about twice as many pages as B^+-Tree to index the same number of records. This may elevate the page reclaiming cost known as the *garbage collection overhead*, thus damaging performance and shortening the lifetime of flash memory.

The page layout of μ-Tree is not space-efficient since the leaf node size does not have to be always half of a page. For example, Fig. 5a depicts a μ-Tree of height three that occupies five flash pages. Dark gray regions denote obsolete or unused nodes. White regions and gray regions indicate free space and the space used by valid nodes, respectively. In many cases, the space usage of index nodes is not dense, as shown in Fig. 5a. This is because the ratio of the child node size to the parent node size is exactly two, while the fanout of a node is much greater than two. As a result, a significant amount of space is wasted in higher level nodes.

If we enlarge the leaf node size as shown in Fig. 5b, we can accommodate the same number of records in fewer pages. However, the use of larger leaf node is not always advantageous, because it will inevitably allocate a smaller space to index nodes, possibly leading to the height growth due to the lack of pointers to leaf nodes. Since the height of a tree has great influence on the latency of tree operations, we should avoid the height growth as much as possible. As there is a tradeoff between space efficiency and performance, the first issue we have to consider is, when the number of records is fixed, how to obtain the best page layout of μ-Tree that can minimize both the number of pages occupied by the tree and the height of the tree.

Although it is possible to find the best layout for the given number of records, it requires some a priori knowledge of the target workload to estimate the number of

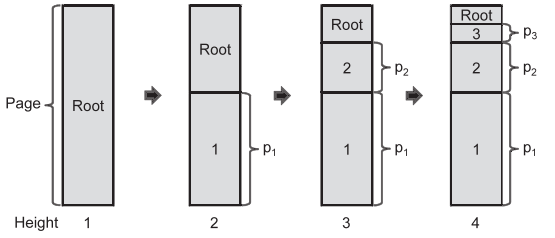


Fig. 6. The generalized page layout in μ^*-Tree.

records it handles. However, the number of records to be indexed by μ-Tree not only varies from workload to workload, but also changes over time dynamically even in the same workload. There is clearly no single page layout that works for all cases. Therefore, it is necessary to develop an adaptive, self-organizing layout scheme that automatically adjusts the page layout according to the current state of the tree on-the-fly. The second issue is to find out when and how the page layout should be adjusted to improve space efficiency and performance.

Whenever the page layout is changed, it is not acceptable if we have to reorganize the entire nodes under the new layout and rewrite them into NAND flash memory. This will cause considerable read and write overhead. Instead, we need to devise a scheme that allows different layouts to coexist in the same tree simultaneously, which is the final issue addressed in this paper.

4 μ^*-Tree

μ^*-Tree is an improved version of μ-Tree. While μ-Tree has static page layout, μ^*-Tree has generalized the page layout so that the node size at each level can be configured flexibly. Furthermore, each page in μ^*-Tree can have its own layout. In μ-Tree, all nodes in the same level have the same size during the lifetime of the tree. In contrast, the page layout evolves in μ^*-Tree according to the current state of the tree. The specific page layout is determined by the proposed adaptive page layout scheme, which helps μ^*-Tree minimize the number of pages occupied by the tree and postpone the height growth as long as possible.

4.1 Layout Analysis

4.1.1 Generalizing μ-Tree Layout

We first generalize the page layout of the previous μ-Tree. μ^*-Tree defines the *proportion value* for each level as the ratio of the node size to the single page size. As shown in Fig. 6, the size of nonroot nodes at level L is determined by the proportion value p_L, and the root node occupies the rest of the page. Like μ-Tree, a leaf node takes up the entire page when the height is one.

Let us denote the height of μ^*-Tree as H, and the proportion value at level L as p_L, where $0 < p_L \leq 1$. In μ^*-Tree, the node size, N_L, at level L is given by

$$N_L = \begin{cases} P & \text{if a single level } (H = 1); \\ P \cdot \left(1 - \sum_{i=1}^{L-1} p_i\right) & \text{if root } (L = H \text{ and } H > 1); \\ P \cdot p_L & \text{otherwise } (L < H \text{ and } H > 1), \end{cases} \quad (2)$$

TABLE 1
Summary of Notations

Symbols	Definitions
p_L	the proportion value at level L
n_I	the upper bound on the total number of pointers in index nodes
n	the number of records
f	the fanout of a single page
H	the height of μ^*-Tree
h	the minimum height of μ^*-Tree for the given n records
c_r , c_w	the read and write flash operation latencies
C_{gc}	the garbage collection cost for a single victim page
C_u	the cost of an update operation for μ^*-Tree
O_{gc}	the number of garbage collection operations triggered
O_u	the number of update operations
e	the total number of pages allocated to μ^*-Tree
P_v	the probability that a victim page has valid leaf node

where P represents the page size of NAND flash memory. Note that the original μ-Tree layout can be seen as an instance of the generalized layout if we set the proportion value $p_L = (\frac{1}{2})^L$, for all $L < H$.

The generalized layout still retains the property of μ-Tree such that nonroot nodes are not influenced by the change in the height. At the same time, the generalized layout allows us to configure the tree more flexibly and in a fine-grained manner. However, the proportion values should be chosen carefully as they have great impact on the overall performance. The larger leaf node lessens space overhead, but makes the tree grow up faster. The smaller leaf node prevents fast growth of the tree, but occupies more pages for the same number of records.

4.1.2 Finding the Best Layout

We mathematically analyze the page layout to find the best layout which, for the given number of records, minimizes the cost of an update (i.e., insert and delete) operation. Table 1 summarizes the notation used in our analysis.

The optimal ratio of index nodes is obtained when the number of pointers to leaf nodes is maximized. When the leaf node size is fixed, the number of pointers in index nodes decides the height of the tree. This is because if there are not enough pointers in index nodes, the height of the tree should be raised to make room for more pointers. Hence, maximizing the number of pointers to leaf nodes is equivalent to minimizing the height of the tree.

The upper bound on the total number of pointers in index nodes, n_I, can be represented as $n_I = fp_2 \times fp_3 \times \cdots \times fp_H$, where f and H denote the fanout of a single page and the height of the tree, respectively. In μ^*-Tree, all nodes in the path from the root node to a leaf node should be stored in a single page, thus, $p_2 + p_3 + \cdots + p_H = 1 - p_1$. Using the AM-GM (Arithmetic Mean-Geometric Mean) inequality,[3] we can find the ratio that maximizes n_I as follows:

$$p_2 = p_3 = \cdots = p_H = \frac{1 - p_1}{H - 1}. \quad (3)$$

Therefore, the same proportion values p_L $(L \geq 2)$ for all index nodes are optimal in that they minimize the height of

3. For any n nonnegative real numbers $x_1, x_2, \ldots, x_n$, the inequality $\frac{x_1+x_2+\cdots+x_n}{n} \geq \sqrt[n]{x_1 x_2 \cdots x_n}$ holds the equality if and only if $x_1 = x_2 = \cdots = x_n$.

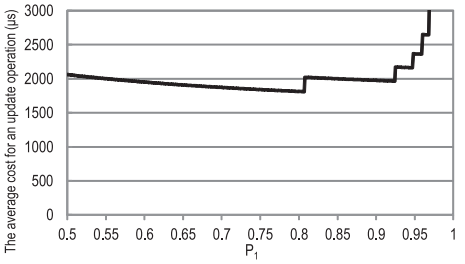


Fig. 7. The average cost for an update operation according to p_1.

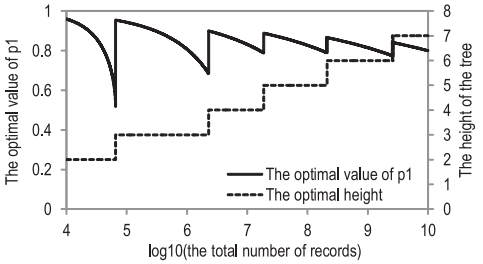


Fig. 8. The optimal value of p_1 and the optimal height according to the number of records.

the tree. Consequently, the maximum number of records, n, that μ^*-Tree of the height H can accommodate is given by

$$n = f^H p_1 \left(\frac{1-p_1}{H-1}\right)^{H-1}. \tag{4}$$

For the given n records, the minimum height of the tree h can be derived from (4) as follows:

$$h = \left\lceil e^{W_{-1}\left(\frac{\log p_1 + \log f - \log n}{e^{\log(1-p_1)+\log f}}\right) + \log(1-p_1) + \log f} \right\rceil + 1, \tag{5}$$

where W_{-1} indicates the Lambert W Function with $k = -1$ [12]. The Lambert W Function cannot be derived in terms of elementary functions, but it can be approximated using iterative algorithms such as Newton's method [13].

The proportion value of leaf nodes, p_1, directly affects the number of pages possessed by μ^*-Tree. As p_1 is getting larger, the number of pages used by the tree gets smaller, and it lessens the garbage collection overhead.

Let us denote by C_{gc} the garbage collection cost for a single victim page. Since valid nodes in the victim page are retrieved and then rewritten into a new page during garbage collection, C_{gc} is essentially the same as the cost of an update operation, C_u, as follows:

$$C_{gc} = C_u = c_r h + c_w, \tag{6}$$

where c_r and c_w indicate the read latency and the write latency for a single page in NAND flash memory, respectively.[4]

Suppose that O_u denotes the number of update operations and e is the total number of pages allocated to μ^*-Tree. Since μ^*-Tree writes one single flash page for each update operation, the number of pages to be reclaimed, N_r, can be represented as follows:

$$N_r = O_u - e. \tag{7}$$

The probability that a victim page has valid leaf node, P_v, is simply calculated as the proportion of the number of records (n) to the total number of records that all leaf nodes can accommodate (efp_1), thus

$$P_v = \frac{n}{efp_1}. \tag{8}$$

Since garbage collection is performed only for valid node, we can derive the total number of garbage collection operations triggered by update operations, O_{gc}, multiplying the number of pages to be reclaimed (N_r) by the probability that a victim page has valid leaf node (P_v), as follows:

$$O_{gc} = N_r P_v = (O_u - e)\frac{n}{efp_1}. \tag{9}$$

Consequently, the objective value Φ, which represents the average cost for an update operation is defined as follows:

$$\Phi = \frac{O_u C_u + O_{gc} C_{gc}}{O_u} = \frac{\left(O_u + \frac{(O_u - e)n}{efp_1}\right)(c_w + c_r h)}{O_u}. \tag{10}$$

We plot Φ varying the value of p_1 from 0.5 to 1, as illustrated in Fig. 7. We set all the proportion values of index nodes identically as suggested in (3). The cost of reading or writing a flash page is modeled after MLC NAND flash memory [8] such that $c_r = 165.6\ \mu$s and $c_w = 905.8\ \mu$s. The other parameter values are set as follows: $n = 1{,}000{,}000$, $e = 8{,}192$, $O_u = 1{,}000{,}000$, and $f = 512$. The value of Φ is minimized when p_1 is 0.81, and suddenly jumps when p_1 is 0.82, 0.93, and 0.95, due to height growth. Within the same height, Φ decreases as the value of p_1 is getting larger because of the reduced garbage collection overhead.

Fig. 8 shows the optimal value of p_1 which minimizes the objective value Φ, and the optimal height of μ^*-Tree, when we vary the total number of records (n) from ten thousands (10^4) to ten billions (10^{10}). Other parameters are configured the same way as in Fig. 7. As n grows, the optimal value of p_1 is decreased to delay the increase in the tree height. When the number of records reaches certain points (such as 6.6×10^4, 2.3×10^6, etc.), p_1 jumps rapidly as the tree height is raised by one. Right after the height is increased, it is optimal to set p_1 as large as possible again to accommodate more number of records within the current height.

In conclusion, when the number of records n is known, the optimal layout of μ^*-Tree can be found as follows: 1) Find p_1 from Fig. 8. This is the value of p_1 which minimizes the update cost Φ in (10). 2) Set the proportion values p_L ($L > 2$) for index nodes identical as shown in (3).

4.1.3 Limit on the Number of Records

Since all nodes along the path from the root to the leaf should be stored in a single page, the maximum height of μ-Tree or μ^*-Tree is determined by the page size, thus limiting the number of records that can be indexed by the tree.

Another advantage of μ^*-Tree over μ-Tree is that μ^*-Tree can index much more number of records. The height of μ-Tree is quickly limited due to the nature of exponential decrease in the node size. When the page size is 4 KB and the entry size is 8 bytes (i.e., $f = 512$), the maximum height of μ-Tree is

4. A block containing invalidated victim pages should be erased during garbage collection, but we ignore this cost because it is very small compared to other factors.

limited to 9, accommodating up to 1.6×10^{11} records only. By contrast, in the same situation, the height of μ^*-Tree can grow up to 256 and it can index up to 1.2×10^{77} records.

4.2 Adaptive Page Layout Scheme

4.2.1 Overview

Although the previous analytical results are helpful to understand the best layout when the number of records is fixed, predicting the number of records to be handled by μ^*-Tree is almost impossible in practice. For this reason, we developed an adaptive page layout scheme for μ^*-Tree.

The adaptive page layout scheme gradually varies the page layout according to the current state of μ^*-Tree on-the-fly. The leaf node size plays an important role in controlling the page layout. Under the proposed scheme, each page can have its own leaf node size if necessary. Whenever we insert a record, the adaptive page layout scheme tries to postpone the height growth by shrinking the leaf node size. On the contrary, it also attempts to enlarge the leaf node size to reduce the number of occupied pages when deleting a record. Once the leaf node size is adjusted, the rest of the page is divided evenly and allocated to index nodes, as suggested in Section 4.1.2.

μ^*-Tree remembers the current layout used by the most recently written page. At the end of each update operation, the current layout may vary gradually if some conditions are satisfied, and only the pages written after the layout variation are affected by the new layout. When a node under the old layout is updated, the current layout is applied to all the updated nodes from the leaf to the root; they are resized according to the current layout and written into a new page.

4.2.2 Conditions for Layout Variation

At the end of each update (i.e., insert and delete) operation, μ^*-Tree checks for certain conditions to see if it needs to change the layout. In case of insertion, the leaf node size shrinks if at least one of the following two conditions is satisfied:

1. when the root node is full,
2. $\frac{Split(I)}{Split(L)} > \frac{1-p_1}{p_1}$,

where $Split(I)$ and $Split(L)$ represent the total number of splits occurred in index nodes and in leaf nodes, respectively.

Recall that μ^*-Tree shrinks the leaf node size to postpone the height growth as long as possible. When the root node is full, the next insertion to the root node will raise the height of the tree. In order to prevent this, we shrink the leaf node size and give more space to index nodes.

When a number of records are inserted into a narrow range of key values, the height of the tree can still grow faster than our expectation. This is because the layout variation is applied only to the updated leaf and index nodes, without changing the page layout of existing nodes belonging to other key values. If we enlarge index nodes before the root node gets full, it will be helpful to slow down the height increase. For this reason, we added the second condition. When the number of splits in index nodes is greater than a certain threshold, μ^*-Tree also shrinks the leaf node size even though the root node is not full. The rapid layout variation is evaded by the enlarged index nodes.

For delete operations, the leaf node size is enlarged only if the utilization of the root node is less than 50 percent. Unlike insertions, we do not vary the layout aggressively for deletions. If we did, it would cancel the layout variation caused by the previous insert operation, incurring unnecessary overhead when insert and delete operations are performed one after the other.

4.2.3 Layout Variation Policy

The page layout of μ^*-Tree is adjusted using three parameters: α, β, and δ. α and β represent the upper bound and the lower bound of the proportion value of the leaf node, respectively, i.e., $\beta \le p_1 \le \alpha$. δ denotes the amount of the proportion value adjusted at a time due to the layout variation.

The adaptive layout scheme is not activated when the height of μ^*-Tree is one as the entire page is allocated to a leaf node. When the height of μ^*-Tree increases from one to two, p_1 is initialized to α. Afterwards, if the conditions for layout variation shown in Section 4.2.2 are satisfied, p_1 is increased or decreased by δ.

The value of p_1 cannot be less than β, or cannot be greater than α. If the root node is full and $p_1 - \delta$ is less than β, the height of μ^*-Tree is raised by one and p_1 is initialized to α again. In contrast, if the root node has only one entry when $p_1 + \delta$ is greater than α, the height is reduced by one with p_1 being initialized to β.

Algorithm 1 summarizes the steps for the proposed adaptive page layout scheme, where $R.entry$ and $R.max$ indicate the number of entries in the root node and the maximum number of entries that can be accommodated in the root node, respectively. The algorithm *AdaptiveLayoutVariation*() is called at the end of each update operation.

Algorithm 1. AdaptiveLayoutVariation

```
1: R ← get the root node
2: C ← get the current layout of μ*-Tree
3: H ← get the current height of μ*-Tree

4: if R.entry = R.max or Split(I)/Split(L) > (1 − C.p1)/C.p1 then
5:   if C.p1 − δ ≥ β then
6:     C.p1 ← C.p1 − δ
7:   else
8:     C.p1 ← α
9:     H ← H + 1
10:  end if
11: else if R.entry < R.max/2 then
12:  if C.p1 + δ ≤ α then
13:    C.p1 ← C.p1 + δ
14:  else if H > 1 then
15:    C.p1 ← β
16:    H ← H − 1
17:  end if
18: end if
19: C.pi ← (1 − C.p1)/(H − 1), for all i ≥ 2
```

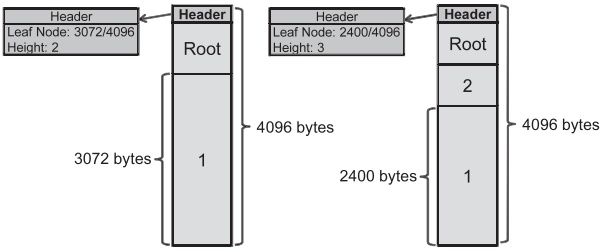


Fig. 9. Page header examples.

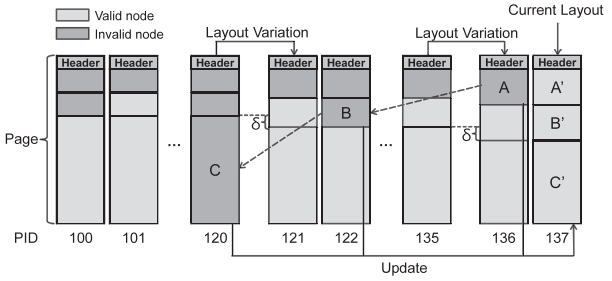


Fig. 10. An example of μ^*-Tree update.

4.3 Implementation of μ^*-Tree

4.3.1 Supporting Different Page Layouts

In μ^*-Tree, the page layout can be dynamically changed after each update operation. The modified layout is applied only to those pages written after the layout variation since rewriting all the previous pages according to the new layout is not practical due to large overhead. This means that μ^*-Tree should provide a mechanism in which flash pages with different page layouts can coexist in the same tree and they can refer to each other freely.

For this reason, each page has a *page header* which stores its layout information. When accessing a node, μ^*-Tree first reads the page header from the page the node belongs to and calculates the offset and the size of the node based on the layout information. Due to page headers, pages with the previous layout do not have to be rewritten and they can be reused regardless of the layout or height variation in μ^*-Tree.

As shown in Fig. 9, the page header is located on top of each page, storing the layout information when the page was written such as the leaf node size and the height of the tree. Because index nodes always have the same size, we can calculate the offset and the size of every index node using these two values.

Fig. 10 illustrates an example of update operation performed in μ^*-Tree of the height three. A page header is located on top of every page and different pages have different layouts in μ^*-Tree. The space for index nodes is evenly divided regardless of the height or layout variation. In Fig. 10, the layout variation occurred when μ^*-Tree writes flash pages 121 and 136. We can see that the leaf node size is reduced by δ after the layout variation. In case the leaf node C is updated, all of its parent nodes, B and A, also should be updated. Except for the root node A, updated nodes B and C have layouts different from the current layout. They are resized to B' and C', respectively, and finally written into the page 137 with the updated root node A'. The node C' may split when all the entries previously stored in C cannot be accommodated in C'.

4.3.2 μ^*-Tree Operations

There are three major differences between μ^*-Tree operations and μ-Tree (or B$^+$-Tree) operations. First, a node can be split in μ^*-Tree not only when a new entry is inserted to the node, but also when the node is updated. In μ-Tree or B$^+$-Tree, a node split occurs only if the node is full and no entry is available for the newly inserted record. In μ^*-Tree, however, a node split also takes place when the node needs to be updated but its node size becomes smaller in the current page layout due to the layout variation. As a result, all operations except for retrieval can cause node splits in μ^*-Tree. Second, a node can be split into more than two nodes in μ^*-Tree, whereas a node is always divided into two nodes in B$^+$-Tree and μ-Tree. For example, if the node size to be split is s_1 and the corresponding node size in the current layout is s_2, the node is split into $\lceil \frac{s1}{s2} \rceil$ nodes in μ^*-Tree. Finally, μ^*-Tree should read the page header to locate the position of each node.

Retrieval. The retrieval process of μ^*-Tree is almost same as that of μ-Tree, since it does not change the page layout. The only difference is that μ^*-Tree is required to calculate the position of each node using the layout information in the page header.

Insertion. The insertion process first finds the target leaf node using the retrieval process. μ^*-Tree inserts a key into the node and allocates a new page to store all updated nodes. For every updated node at each level, μ^*-Tree compares its layout with the current layout. If two layouts do not match, μ^*-Tree changes the node size so that it fits into the current layout. And then, all updated nodes are written into the new page. After a record is inserted, the page layout can be changed if the condition described in Section 4.2.2 is satisfied.

If a node overflows during the insertion process, μ^*-Tree splits the node into k ($k \geq 2$) nodes and prepares the total k flash pages by allocating the additional $k-1$ pages. And μ^*-Tree writes split nodes into k pages sequentially. The parent nodes are written to the last page with the last split node. Splitting nodes can take place recursively for each level, and the height of μ^*-Tree grows up if the root node needs to be split.

Deletion. The deletion process is almost same as the insertion process. μ^*-Tree deletes a key from the target leaf node, and allocates a new page to store all updated nodes. And then, it compares the layout of updated nodes with the current layout, modifies the node size if necessary, and writes all the updated nodes into the new page. Finally, μ^*-Tree checks for the condition for possible layout change.

When a node is resized according to the current layout, an overflow can occur. In this case, the node is split into more than two nodes in the same way as in insertion. If the overflowed node is the root node, the height of μ^*-Tree is increased. On the other hand, the updated node is deleted if it has no entry. If the deleted node is the root node, the height of μ^*-Tree is decreased.

As in μ-Tree, μ^*-Tree does not perform balancing or redistribution which may incur additional flash access overhead. In fact, the adaptive page layout scheme serves as a substitute for such mechanisms.

TABLE 2
The Characteristics of Traces

Trace Name	Description	Retrievals	Updates	# Records
kernel	Tree operations in ReiserFS obtained while untar, compile, clean, and remove the Linux 2.6.16 kernel source.	2,274,867	497,001	95,527
postmark	Tree operations in ReiserFS obtained from the Postmark benchmark that models an e-mail server. Up to 100,000 files from 512 bytes to 10 KB are created and deleted.	4,617,494	965,995	286,182
tpcc	Tree operations in μ-FTL obtained from the TPC-C benchmark with 16 warehouses.	5,352,317	4,647,683	264,203
financial	Tree operations in μ-FTL obtained from OLTP (OnLine Transaction Processing) environment.	4,735,981	5,264,019	116,058
general	Tree operations in μ-FTL obtained from a 5-day long general laptop usage on Windows XP including office works, download, etc.	2,269,097	2,730,903	104,662
web	Tree operations in μ-FTL obtained from one-day long laptop web surfing on Windows XP.	2,843,611	2,156,389	109,576

4.3.3 In-Memory Cache

Most of previous flash-aware index schemes such as BFTL [14], FD-tree [15], and LA-tree [16] depend on in-memory structures to temporarily cache tree updates. Such an in-memory cache delays the actual flash accesses and batches incoming updates together, thereby reducing the number of flash operations. By contrast, the performance improvement of μ^*-Tree over B^+-Tree is basically originated from its page layout. In particular, μ^*-Tree does not rely on any in-memory structures such as mapping table and node cache to operate. This means that the benefit of the adaptive page layout scheme used in μ^*-Tree is orthogonal to the use of an in-memory cache. Consequently, we can expect that even though we use an in-memory cache for μ^*-Tree, it will still outperform B^+-Tree with the same cache.

To verify this, we implement a simple in-memory cache that can be used in both μ^*-Tree and B^+-Tree. The memory space is split into two partitions: operation buffer and node cache. The operation buffer temporarily holds incoming requests to absorb and coalesce actual tree operations to μ^*-Tree or B^+-Tree. All operations first pass through this buffer, and go to the tree on cache misses. On a cache hit, i.e., when the previous operation on the same key exists in the operation buffer, the corresponding value in the buffer is either updated (for write operation) or returned (for read operation), and the operation terminates without triggering actual tree operation. The node cache keeps the nodes accessed by previous operations so that subsequent tree operations on the same node can be performed without any flash operations. Both the operation buffer and the node cache are managed in LRU fashion. We allocate the half of the configured memory for the operation buffer and the other half for the node cache.

5 Evaluation

5.1 Evaluation Methodology

μ^*-Tree is built on a trace-driven NAND flash memory simulator which performs raw flash read, write, and erase operations. The simulator has an ability to count the number of operations performed, and these counts are converted to the elapsed time using the operational latency of MLC NAND flash memory: $165.6\ \mu s$ for read, $905.8\ \mu s$ for write, and $1{,}500\ \mu s$ for erase [8]. The simulator can be configured with parameters such as the total capacity of NAND flash memory, the page size, and the number of pages in a block. We configure it with typical parameters of MLC NAND flash memory, where the page is 4,096 bytes in size and each block has 128 pages. The size of a key-pointer pair is set to 8 bytes in all cases. Unless otherwise stated, the parameter values of α, β, and δ for μ^*-Tree are set to 0.9, 0.5, and 1/256 (i.e., 16 bytes or two entries), respectively, throughout the experiment. The garbage collection is triggered when the number of free blocks is less than 10 percent of the total number of blocks, and the victim block is selected using the round-robin policy. For fair comparison, we have also implemented B^+-Tree and μ-Tree on the same simulator.

We have used six traces obtained from real workloads for performance evaluation. Table 2 summarizes the characteristics of traces used in this paper. kernel and postmark are obtained from tree operations of the Reiser file system (ReiserFS), and tpcc, financial, general, and web are from tree operations of μ-FTL [1]. ReiserFS is one of the representative disk-based file systems in Linux which uses B^+-Tree for metadata indexing. μ-FTL is a flash translation layer that supports multiple mapping granularities using variable-sized extents. μ-FTL uses μ-Tree [11] for storing its mapping information. We have also used several microbenchmarks to explore the various aspects of μ^*-Tree. The main performance metric we use is the elapsed time needed for flash memory operations. In cases of the experiments using randomly generated keys, we repeat the same set of experiments for five times, and take the average of three values discarding the maximum and the minimum values.

5.2 Microbenchmarks Results

5.2.1 Adaptive Layout Variation

First, we verify that μ^*-Tree adjusts the page layout adaptively according to the number of records. We insert randomly generated key-record pairs into μ^*-Tree and investigate the change in the leaf node size.

Fig. 11 shows the variation of leaf node size when the number of records is increased from ten thousands to one million. The y-axis represents the proportion value of the leaf node (p_1) and the number of records is displayed in logarithmic scale. The dotted line plots the optimal size of the leaf node obtained from Fig. 8, and the solid line is the actual result. We can see that the leaf node size is initialized to α and it is gradually decreased down to β as the number of records increases. The sudden jump indicates the point where the height of μ^*-Tree is raised. After the height growth, the leaf node size is reset to α and gradually shrinks again.

We observe there is a slight difference between the analytical model and the actual result. This is because the

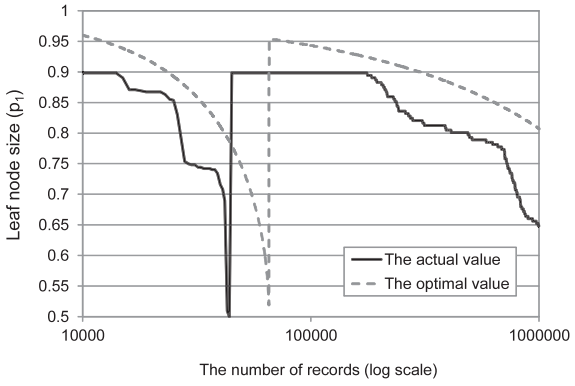


Fig. 11. The variation of the leaf node size in μ^*-Tree.

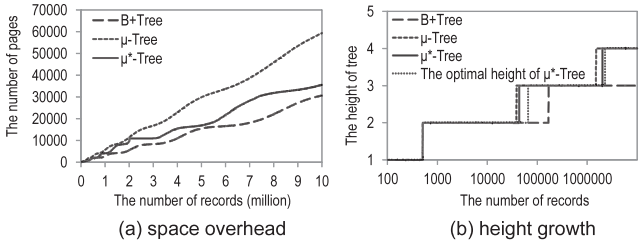


Fig. 12. Comparison of B^+-Tree, μ-Tree, and μ^*-Tree.

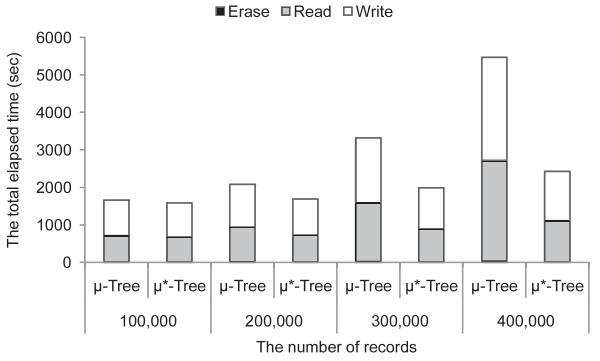


Fig. 13. The total elapsed time for one million update operations.

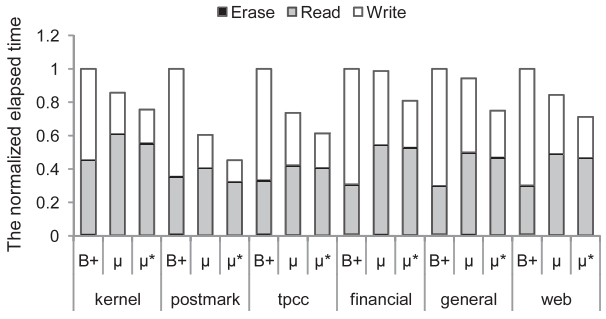


Fig. 14. The normalized elapsed time for real workload traces.

analytical model is based on the assumption that every node is always fully utilized, thus the height of μ^*-Tree is raised only when all nodes are full. This is the ideal case of balanced search trees. In practice, however, the height is increased due to recursive splits from a leaf node to the root node, even though other nodes are not full. Therefore, the actual leaf node size is smaller than the estimated value of the analytical model and the height is, in reality, increased faster than in the analytical model.

5.2.2 Space Overhead

To compare the space overhead of B^+-Tree, μ-Tree, and μ^*-Tree, we insert ten million randomly generated key-record pairs and measure the number of pages occupied by each tree. Fig. 12a illustrates the results. The dashed line, the dotted line, and the solid line denote the results of B^+-Tree, μ-Tree, and μ^*-Tree, respectively.

From Fig. 12, we confirm that the number of pages used by μ-Tree is almost doubled compared to B^+-Tree. The space overhead of μ^*-Tree is slightly larger than B^+-Tree but notably smaller than μ-Tree. In fact, as the leaf node size gets smaller, the number of pages occupied by μ^*-Tree becomes closer to μ-Tree. In the opposite case where the leaf node size gets larger, the number of pages used by μ^*-Tree moves toward B^+-Tree.

5.2.3 Tree Height

Fig. 12b depicts the change in the height when we insert the same randomly generated ten million entries to B^+-Tree, μ-Tree, and μ^*-Tree. The optimal height from Fig. 8 is also illustrated in Fig. 12b for comparison. We can see that the height growth of B^+-Tree is much slower than μ-Tree. This is because B^+-Tree can accommodate more number of records than μ-Tree in the same height since the fanout of a node in B^+-Tree is larger than that in μ-Tree. The height growth of μ^*-Tree is slightly slower than μ-Tree, even though μ^*-Tree uses significantly fewer number of pages. This is due to the adaptive page layout scheme which postpones the height growth as long as possible. Another reason is that the maximum number of pointers in index nodes of μ^*-Tree is larger than that of μ-Tree since μ^*-Tree always evenly divide the space for index nodes while the parent index node size is reduced by half in μ-Tree.

5.2.4 Update Performance

The number of records inserted into the tree has great influence on the update performance. As the number of records increases, the number of pages occupied by the tree also grows, thus garbage collection is invoked more frequently to reclaim obsolete pages.

To compare the update performance of μ-Tree and μ^*-Tree, we first initialize each tree with the given number of records ranging from 100,000 to 400,000. Then, we perform one million update (insert and delete) operations. The space allocated for each tree is configured to 16 MB.

Fig. 13 compares the total elapsed time to complete all the update operations in μ-Tree and μ^*-Tree. Obviously, the total elapsed time increases as there are more number of records in the tree. However, the total elapsed time of μ-Tree grows faster than that of μ^*-Tree. μ^*-Tree outperforms μ-Tree by 56 percent when the number of records is 400,000. This is because the number of pages occupied by μ^*-Tree is much less than that of μ-Tree, improving the garbage collection overhead of μ^*-Tree.

5.3 Real Workloads Results

5.3.1 Overall Performance

Fig. 14 compares the total elapsed time of B^+-Tree, μ-Tree, and μ^*-Tree for the real workload traces. The total elapsed time is normalized to the results of B^+-Tree. In this

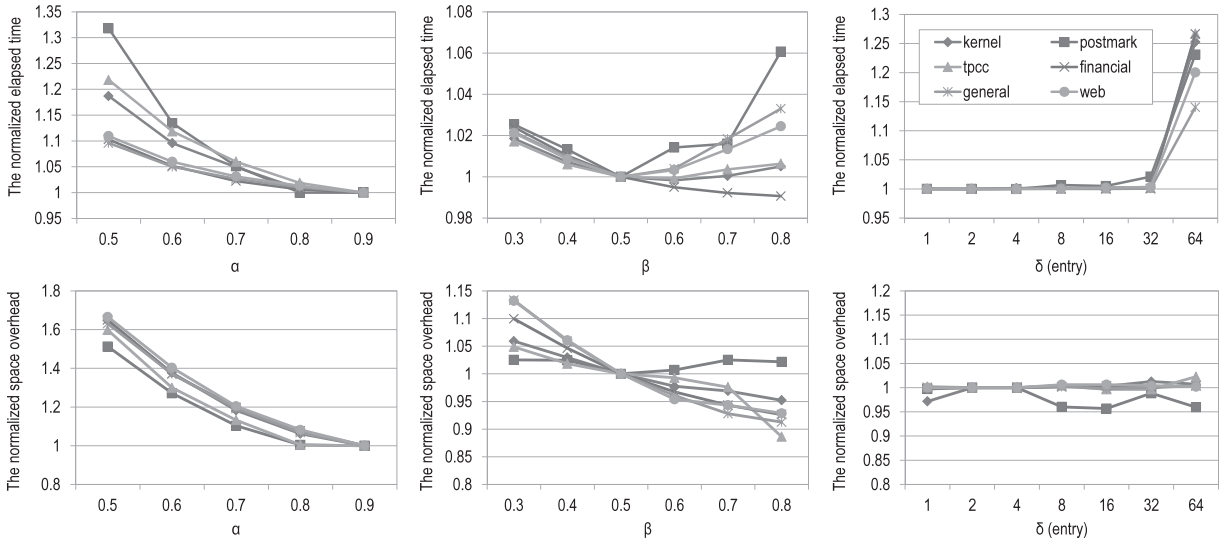


Fig. 15. The total elapsed time and the overall space overhead of μ^*-Tree varying parameter values of α, β, and δ.

experiment, the space allocated to each tree is 16 MB for kernel and postmark, and 64 MB for tpcc, financial, general, and web. This is because the latter four traces have lots of update operations compared to former traces.

In Fig. 14, we observe that μ^*-Tree outperforms B^+-Tree by up to 55 percent for the postmark trace and by 34 percent on average (25 percent for kernel, 39 percent for tpcc, 29 percent for financial, 25 percent for general, and 29 percent for web). Note that μ-Tree has marginal benefits over B^+-Tree for financial and general traces. This is due to the fact that the height of μ-Tree is three while that of B^+-Tree is two. The difference in the height increases the read cost significantly, and it offsets the benefit of the reduced write cost.

μ^*-Tree also outperforms μ-Tree by up to 25 percent for the postmark trace and by 18 percent on average (12 percent for kernel, 17 percent for tpcc, 18 percent for financial, 20 percent for general, and 15 percent for web). Compared to μ-Tree, we can see that the write cost is substantially improved due to the space efficiency of μ^*-Tree.

5.3.2 The Effect of Parameters

To study the effect of parameters used in μ^*-Tree, we evaluate the performance and the overall space overhead of μ^*-Tree varying the parameter values of α, β, and δ one at a time. Fig. 15 summarizes the total elapsed time and the total number of pages occupied by μ^*-Tree when only one parameter value is changed while the others are fixed. The results are normalized to those of the given workload with the default parameter values. The default values of α, β, and δ are 0.9, 0.5, and 1/256, respectively.

When $\alpha = 0.5$, μ^*-Tree works almost same as μ-Tree since the adaptive scheme does not work ($\alpha = \beta$). As the value of α increases, the total elapsed time and the overall space overhead is steadily reduced for all workloads. This is because as α gets larger, the leaf node also has more chance of getting enlarged. However, α cannot be increased beyond a certain value since the fanout of index nodes should be able to accommodate new entries created by the split procedure when the height is increased. These factors should be considered when choosing the value of α.

μ^*-Tree shows the best performance when β equals to 0.5 in most workloads. When $\beta < 0.5$, the leaf node size is smaller than half of a single page. This will make μ^*-Tree require more number of pages and eventually incur more garbage collection overhead. When $\beta > 0.5$, the minimum leaf node size is increased, but the maximum size of index nodes becomes smaller at the same time. Although the space overhead decreases, smaller index nodes accelerate splits in index nodes and make the height of μ^*-Tree grow faster. The financial trace is an exception where its performance is improved constantly as β is increased up to 0.8. It was helpful for the financial trace to have higher page utilization rather than to use the adaptive layout scheme. Yet, the performance difference is not significant.

In case of δ, the smaller value of δ makes the layout variation be fine grained, but the overhead may increase due to frequent layout changes. By contrast, the larger value of δ may degrade the performance due to the coarse-grained layout variation. Fig. 15 shows that the performance degradation is severe when the value of δ is larger than 1/16 (i.e., 256 bytes or 32 entries). However, when the value of δ is less than 1/16, it hardly affects the overall performance. Note that the space overhead is not influenced by the value of δ since it does not directly affect the leaf node size.

5.3.3 The Effect of Adaptive Page Layout Scheme

In order to assess the effect of the proposed adaptive scheme, we evaluate the performance of *static* μ^*-Tree where the adaptive page layout scheme is disabled by configuring α equal to β. Fig. 16 illustrates the total elapsed time for three real workloads (postmark, tpcc, and financial), and three randomly synthesized microbenchmark workloads (micro100k, micro600k, and micro1m), varying the leaf node size from 0.5 to 0.9 statically (i.e., $p_1 = \alpha = \beta$). In the synthesized workloads, one million update operations are performed after μ^*-Tree is initialized with 100,000 records (micro100k), 600,000 records (micro600k), or 1,000,000 records (micro1m). The results are normalized to those of the previous μ^*-Tree with the adaptive page layout scheme enabled.

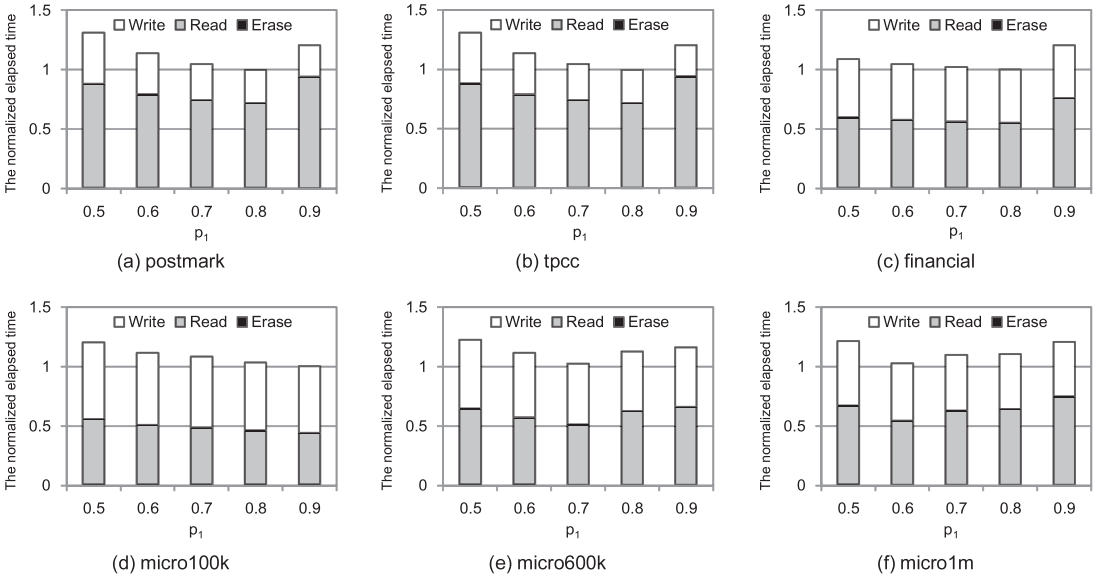


Fig. 16. The total elapsed time for μ^*-Tree after disabling the adaptive page layout scheme according to the value of p_1.

As p_1 is getting larger, the total elapsed time gradually decreases due to the reduced garbage collection overhead. Each workload has its own optimal value of p_1 which minimizes the total elapsed time: 0.8 for real workloads, 0.9 for micro100k, 0.7 for micro600k, and 0.6 for micro1m. If p_1 is enlarged beyond this optimal point, the elapsed time rapidly increases since the height of μ^*-Tree grows up. For all workloads, the proposed adaptive page layout scheme shows comparable or better performance than the best possible result obtained by μ^*-Tree whose page layout is statically configured.

5.3.4 The Effect of In-Memory Cache

Fig. 17 illustrates the total elapsed time of μ^*-Tree for real workloads varying the size of cache from 0 to 256 KB. All results are normalized to those of μ^*-Tree without cache (i.e., 0 KB cache size). As the cache size getting larger, the elapsed time decreases because the cache absorbs actual flash operations. The cache effect of tpcc is relatively small compared to other workloads since its working set is large and access patterns are randomly distributed.

Fig. 18 compares the total elapsed time of μ^*-Tree with that of B^+-Tree using the same in-memory cache mechanism. The cache sizes of both trees are configured to 32 KB, and the results are normalized to those of B^+-Tree. Even though the same caching scheme is used, μ^*-Tree still outperforms B^+-Tree by up to five times. This suggests that μ^*-Tree's adaptive page layout scheme is quite effective regardless of whether there is an in-memory cache or not. In fact, the use of cache does not weaken the advantages of the adaptive page layout scheme in μ^*-Tree.

5.4 Comparison with BFTL

We compare the update performance of μ^*-Tree with that of BFTL [14], which is a popular B^+-Tree layer for flash memory working on top of FTL. BFTL absorbs incoming tree updates in in-memory cache called *reservation buffer*, and flushes them at once. The update records are written on flash pages in a log-structured manner, and BFTL maintains a mapping table called *Node Translation Table* (NTT) to keep track of the scattered records for each node.

Since BFTL works on top of FTL, its performance depends on the underlying FTL. In this evaluation, BFTL is implemented on two kinds of FTLs: DAC [3], which is a representative page mapping FTL and FAST [2], which is a log-based block mapping FTL. We observe the performance of each scheme serving one million random update operations after initializing it with the given number of records. The size of the reservation buffer and the maximum length of NTT lists are configured to 60 and 3, respectively, as recommended in the BFTL paper [14]. For fair comparison, the cache size of μ^*-Tree is set to 128 KB which is almost the same amount of total memory used in BFTL over FAST. We allocate 16 MB for flash space.

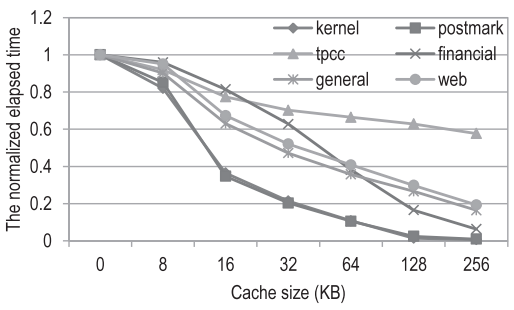


Fig. 17. The total elapsed time of μ^*-Tree varying the size of cache.

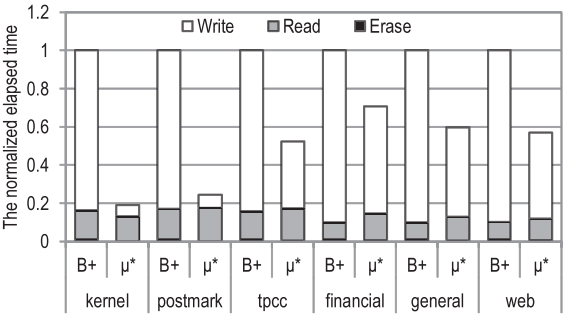


Fig. 18. Comparison of B^+-Tree and μ^*-Tree with 32 KB cache.

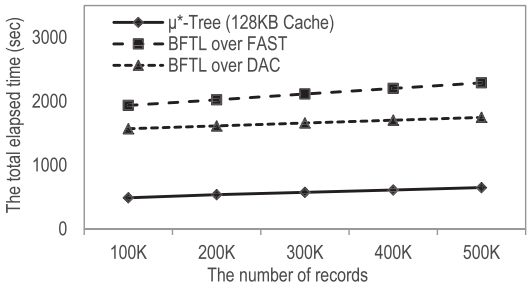


Fig. 19. The total elapsed time for BFTL and μ^*-Tree performing one million update operations.

Fig. 19 shows the total elapsed time for μ^*-Tree and BFTLs. μ^*-Tree outperforms BFTL over FAST by up to four times, and BFTL over DAC by up to 3.2 times. Although BFTL can coalesce update operations, scattered nodes should be merged because NTT cannot grow unlimitedly. The node merge operation occurs very frequently under random updates, triggering a large amount of flash read and write operations. Furthermore, invalid logs should be compacted to gather free space while garbage collector in the underlying FTL also reclaims invalid flash pages at the same time. This duplicated garbage collection degrades the performance. By contrast, since μ^*-Tree manages mapping and garbage collection on raw flash by itself, the overall overhead can be much smaller compared to BFTL and other similar approaches that work on FTL.

6 Related Work

A number of schemes have been proposed for developing efficient index structures for flash memory. BFTL [14] is one of the most popular B^+-Tree layer for flash memory, making tradeoff between read and write performance. BFTL maintains in-memory pool to buffer newly created or modified nodes, and those buffered nodes are flushed to log pages in flash memory periodically or when the in-memory pool becomes full. Although BFTL can improve the write performance, scattered log pages may degrade the read performance.

FlashDB [4] addresses this problem by employing an adaptive algorithm which switches two separate modes: *LogMode* and *DiskMode*. LogMode handles write-intensive nodes such as index nodes by writing them to a list of log pages similar to BFTL. On the other hand, read-intensive nodes such as leaf nodes are written directly into flash memory as a single page in DiskMode. This algorithm can improve the write performance without suffering from poor read performance caused by scattered leaf nodes.

Recently, FD-Tree [15] is introduced to lessen the small random write overhead. FD-Tree consists of multiple levels of fractional sorted runs, and a small in-memory B^+-Tree called head tree on top level. Updates are stored in head tree first, and merged into lower level sorted runs subsequently. FD-Tree avoids random writes by rewriting merged runs into flash pages in batch.

All of the aforementioned index structures are implemented on top of FTL, hence they do not consider important flash memory management issues such as out-of-place update and garbage collection. Since index structures are not aware of flash page allocation and mapping, the chance of improvement is limited. By contrast, μ^*-Tree deals with raw flash pages directly without the help of intermediate layers such as FTL. Since μ^*-Tree itself manages flash pages and garbage collection, the overall performance and space efficiency can be improved further. We believe μ^*-Tree is an essential component in building flash-aware DBMSes and file systems.

MicroHash [17] is a hash-based index structure that also works on raw flash memory. MircoHash is developed for flash-based wireless sensor devices to obtain both high performance and low energy consumption, but not appropriate for a general index structure in file systems or DBMSes due to its scalability problem.

Agrawal et al. have suggested Lazy Adaptive Tree (LA-Tree) [16], which is designed to optimize accesses to raw flash memory. A number of fixed-size subtrees based on B^+-Tree at each level organizes the entire tree, and one buffer is attached to each subtree to batch updates on the tree and its descendants. These buffers are lazily merged with the tree at an appropriate time. LA-Tree uses an online adaptive algorithm, which decides the optimal time to merge the buffers considering tradeoff between update performance and lookup performance. As mentioned in LA-tree paper [16], the approaches of LA-Tree and μ^*-Tree are orthogonal because the objective of LA-Tree is minimizing the number of accesses to flash while that of μ^*-Tree is reducing the overhead associated with an update operation.

7 Conclusions

An index structure natively works on NAND flash memory is essential for many flash-aware file systems, flash-aware DBMSes, and FTLs. Therefore, the importance of a space-efficient index structure cannot be overemphasized especially for resource-constrained embedded systems. This paper proposes an index structure called μ^*-Tree, which aims at improving performance over B^+-Tree.

μ^*-Tree can handle tree update requests efficiently on NAND flash memory by storing all the updated nodes into a single flash page. μ^*-Tree has generalized the page layout so that the node size can be configured in a fine-grained manner. Under the generalized page layout, we show that, for the given number of records, it is possible to find the best page layout which maximizes space efficiency and performance through analytical modeling.

However, as the number of records to be indexed by μ^*-Tree cannot be predicted in practice, we propose an adaptive page layout scheme which dynamically adjusts the page layout at runtime. According to the current state of μ^*-Tree, the adaptive page layout scheme enlarges or shrinks the leaf node size to accommodate more number of records within the current height. The rest of a page is evenly divided and then allocated to index nodes to maximize the number of pointers to leaf nodes. The changed page layout is effective only for those pages written after the layout variation. μ^*-Tree allows flash pages with different page layouts to coexist in the same tree by maintaining a page header in each page.

Our evaluation results with the real workload traces show that μ^*-Tree outperforms B^+-Tree by up to 55 percent in terms of the total elapsed time. With a small in-memory cache of 32 KB, μ^*-Tree improves the overall performance

by up to five times compared to B^+-Tree with the same cache size. In the near future, we plan to use μ^*-Tree for developing flash-aware file systems and extent-based FTLs.

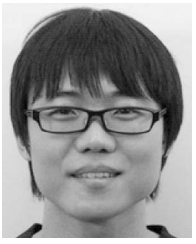
ACKNOWLEDGMENTS

This work was supported by Next-Generation Information Computing Development Program (No. 2011-0020520) and by Mid-career Researcher Program (No. 2011-0027613) through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology. This work was also partly supported by the IT R&D program of MKE/KEIT (KI10041244, SmartTV 2.0 Software Platform).

REFERENCES

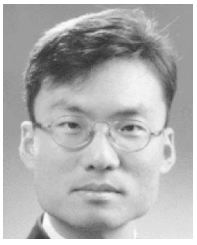
[1] Y.-G. Lee, D. Jung, D. Kang, and J.-S. Kim, "μ-ftl: A Memory-Efficient Flash Translation Layer Supporting Multiple Mapping Granularities," *Proc. Int'l Conf. Embedded System Software (EMSOFT),* 2008.
[2] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation," *ACM Trans. Embedded Computer Systems,* vol. 6, no. 3, p. 18, 2007.
[3] M.-L. Chiang, P.C.H. Lee, and R.-C. Chang, "Using Data Clustering to Improve Cleaning Performance for Flash Memory," *Software Practice and Experience,* vol. 29, no. 3, pp. 267-290, 1999.
[4] S. Nath and A. Kansal, "FlashDB: Dynamic Self-Tuning Database for NAND Flash," *Proc. Int'l Conf. Information Processing in Sensor Networks,* 2007.
[5] G.J. Kim, S.C. Baek, H.S. Lee, H.D. Lee, and M.J. Joe, "LGeDBMS: A Small DBMS for Embedded System with Flash Memory," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2006.
[6] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica,* vol. 1, no. 3, pp. 173-189, 1972.
[7] D. Comer, "Ubiquitous B-Tree," *ACM Computing Surveys,* vol. 11, no. 2, pp. 121-137, 1979.
[8] Samsung, "2gx8 Bit Nand Flash Memory (k9gag08u0m-p)," Samsung Electronics, 2009.
[9] Samsung, "2gx8 bit Nand Flash Memory (k9wag08u1a)," Samsung Electronics, 2009.
[10] A. Silberschatz, H.F. Korth, and S. Sudarshan, *Database System Concepts.* McGraw-Hill, 1997.
[11] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim, "μ-Tree: An Ordered Index Structure for Nand Flash Memory," *Proc. Int'l Conf. Embedded System Software (EMSOFT),* 2007.
[12] R.M. Corless, G.H. Gonnet, D.E.G. Hare, D.J. Jeffrey, and D.E. Knuth, "On the Lambert W Function," *Advances in Computational Math.,* vol. 5, pp. 329-359, 1996.
[13] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes - The Art of Scientific Computing.* Cambridge Univ. Press, 1992.
[14] C. Wu, T. Kuo, and L. Chang, "An Efficient B-tree Layer Implementation for Flash-Memory Storage Systems," *ACM Trans. Embedded Computing Systems,* vol. 6, no. 3, article 19, 2007.
[15] Y. Li, B. He, Q. Luo, and K. Yi, "Tree Indexing on Flash Disks," *Proc. IEEE 25th Int'l Conf. Data Eng. (ICDE),* 2009.
[16] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh, "Lazy-adaptive Tree: An Optimized Index Structure for Flash Devices," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2009.
[17] S. Lin, D. Zeinalipour-Yazti, V. Kalogeraki, D. Gunopulos, and W.A. Najjar, "Efficient Indexing Data Structures for Flash-Based Sensor Devices," *ACM Trans. Storage,* vol. 2, no. 4, pp. 468-503, 2006.

Jung-Sang Ahn received the BS and MS degrees in computer science from KAIST in 2008 and 2010, respectively. Currently, he is working toward the PhD degree in computer science at the same school. His research interests include flash memory, embedded systems, and operating systems.



Dongwon Kang received the BS and MS degrees in computer science from KAIST in 2006 and 2008, respectively. He is currently with Google Korea LLC as a software engineer. His research interests include operating systems and storage systems.

Dawoon Jung received the BS, MS, and the PhD degrees in computer science from KAIST in 2002, 2004, and 2009, respectively. He is currently a senior research engineer in memory division, Samsung Electronics Corporation. His research interests include operating systems and storage systems.



Jin-Soo Kim received the BS, MS, and PhD degrees in computer engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He is currently an associate professor in Sungkyunkwan University. Before joining Sungkyunkwan University, he was an associate professor in KAIST from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of research staff, and with the IBM T.J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems. He is a member of the IEEE and the IEEE Computer Society.

Seungryoul Maeng received the BS degree in electronics engineering from Seoul National University (SNU), Korea, in 1977, and the MS and PhD degrees in computer science in 1979 and 1984, respectively, from KAIST, where he has been a faculty member in the Department of Computer Science since 1984. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar. His research interests include microarchitecture, parallel computer architecture, cluster computing, and embedded systems.