μ-Tree : An Ordered Index Structure for NAND Flash Memory*

Dongwon Kang Dawoon Jung Jeong-Uk Kang Jin-Soo Kim
Computer Science Division
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon 305-701, Korea
{dwkang,dwjung,ux}@camars.kaist.ac.kr jinsoo@cs.kaist.ac.kr

ABSTRACT

As NAND flash memory becomes increasingly popular as data storage for embedded systems, many file systems and database management systems are being built on it. They require an efficient index structure to locate a particular item quickly from a huge amount of directory entries or database records. This paper proposes μ-Tree, a new ordered index structure tailored to the characteristics of NAND flash memory. μ-Tree is a balanced tree similar to B^+-Tree. In μ-Tree, however, all the nodes along the path from the root to the leaf are put together into a single flash memory page in order to minimize the number of flash write operations when a leaf node is updated. Our experimental evaluation shows that μ-Tree outperforms B^+-Tree by up to 28% for traces extracted from real workloads. With a small in-memory cache of 8 Kbytes, μ-Tree improves the overall performance by up to 90% compared to B^+-Tree with the same cache size.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing methods; D.4.3 [**File Systems Management**]: Directory structures

General Terms

Algorithms, Design, Performance

Keywords

B^+-Tree, NAND Flash, index structure

*This research was supported by the MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment) (IITA-2006-C1090-0603-0020).

EMSOFT'07, September 30–October 3, 2007, Salzburg, Austria.

1. INTRODUCTION

Flash memory is being widely adopted as a storage medium for many portable embedded devices such as PMPs (portable media players), PDAs (personal digital assistants), digital cameras and camcorders, and cellular phones. This is mainly due to the inherent advantageous features of flash memory: non-volatility, small and lightweight form factor, low-power consumption, and solid state reliability.

Flash memory comes in two flavors. The NOR type is usually used for storing codes since it can be directly addressable by processors. On the other hand, the NAND type is accessed on a page basis (typically 512 bytes $\sim$ 4 Kbytes) and provides higher cell densities. The NAND type is primarily used for removable flash cards, USB thumb drives, and internal data storage in portable devices.

As the NAND flash technology development continues to double density growth on an average of every 12 months [23], the capacity of a single NAND chip is getting larger at an increasingly lower cost. The declining cost of NAND flash memory has made it a viable and economically attractive alternative to hard disk drives especially in portable embedded systems. As a result, many flash-aware file systems and embedded database management systems (DBMSs) are currently being built on NAND flash memory [2, 7, 9, 13, 24].

Any file system or DBMS requires an efficient index structure to locate a particular item quickly from a huge amount of directory entries or database records. For small scale systems, the index information can be kept in main memory. For example, JFFS2 keeps the whole index structures in memory that are necessary to find the latest file data on flash memory [24]. Apparently, this approach is not scalable to a larger number of files since memory is a precious resource whose capacity should be minimized in embedded systems. Hence, the index information is usually retained on storage in an organized way.

For decades, many different kinds of index structures have been developed for disk-based storage. Among them, B^+-Tree is one of the most popular index structures for file systems and DBMSs [3]. B^+-Tree is a balanced search tree for sorted data that allows for efficient retrieval, insertion, and deletion of records, each of which is identified by a key. B^+-Tree makes use of block-oriented storage efficiently by keeping related records on the same block. Update and search operations affect only a few storage blocks, thus minimizing the number of storage accesses.

Although B^+-Tree has been successful as an index structure for disk-based file systems and DBMSs, it poses a se-

rious problem for NAND flash-based storage. The problem is due to the unique characteristics of NAND flash memory that does not support in-place update. A *page* is a unit of read and write operations in NAND flash memory which is usually a multiple of hard disk sector size. Once a page is written, it should be erased before the subsequent write operation is performed on the same page. Since the erase operation is performed in a unit of much larger *block* and the operation has much longer latency than read or write operations, it is unacceptable to erase the whole erase block whenever a portion of the block is updated.

The naive implementation of B^+-Tree on NAND flash memory is to use each page as a node for storing keys and records. In B^+-Tree, only keys and pointers to children in the next level are stored in interior nodes and all records are stored at the lowest level of the tree. If an update occurs on a record in the leaf node, the modified leaf node needs to be written at a new empty page due to the *erase-before-write* characteristics of NAND flash memory. Since the physical location of the node has been changed, the pointer in the parent node also needs to be modified, which requires another flash write operation. This situation continues until all the nodes from the leaf to the root are written into NAND flash memory. Consequently, the naive implementation generates as many write operations as the height of B^+-Tree whenever a single record is updated.

In this paper, we propose μ-Tree (mu-Tree or *m*inimally *u*pdated-Tree), a new ordered index structure tailored to the characteristics of NAND flash memory. μ-Tree is a balanced search tree similar to B^+-Tree. In μ-Tree, however, all the nodes along the path from the root to the leaf are put together into a single NAND flash memory page in order to minimize the number of flash write operations when a record is updated, inserted, or deleted. Our experimental evaluation shows that μ-Tree outperforms B^+-Tree by up to 28% for traces extracted from real workloads. With a small in-memory cache of 8 Kbytes, μ-Tree improves the overall performance by up to 90% compared to B^+-Tree with the same cache size.

The rest of the paper is organized as follows. Section 2 discusses the characteristics of NAND flash memory and the problem of the existing index structure. Section 3 describes the proposed index structure. The behavior of μ-Tree is mathematically analyzed in Section 4. Section 5 presents several implementation issues of μ-Tree on NAND flash memory. We present the performance of μ-Tree in Section 6 and compare μ-Tree to other approaches in Section 7. Finally, we conclude in Section 8.

2. BACKGROUND AND MOTIVATION

2.1 NAND Flash Memory Characteristics

A NAND flash memory chip is composed of a fixed number of blocks[1], where each block has a number of pages. Each page in turn consists of main data area and spare area. A page is a unit of read and write operations, while a block is a unit of erase operations. The spare area in each page is often used to store error correction code (ECC) and other management information.

[1]The *block* should not be confused with the unit of I/O used in the kernel. Unless otherwise stated explicitly, this paper uses the term *block* to denote the unit of erase operation.

Table 1: The characteristics of SLC [21] and MLC [20] NAND flash memory

	SLC NAND (large block)	MLC NAND
page size	(2K+64)B	(4K+128)B
block size	(128+4)KB	(512+16)KB
# pages / block	64	128
NOP	4	1
read latency	77.8μs (2KB)	165.6μs (4KB)
write latency	252.8μs (2KB)	905.8μs (4KB)
erase latency	1500μs (128KB)	1500μs (512KB)

As the NAND flash memory technology advances, different types of NAND flash memory have been introduced. The first *small block SLC (Single-Level Cell) NAND* has an organization that a block consists of 32 pages and the size of a page is (512 + 16) bytes including 16 bytes of spare data. The next generation of NAND flash memory called the *large block SLC NAND* provides higher capacity. In the large block SLC NAND, the number of pages in a block is doubled and the page size is increased by 4 times compared to the small block SLC NAND. The small block SLC NAND is being phased out of the market in favor of the large block NAND.

The recently introduced *MLC (Multi-Level Cell) NAND* uses an architecture that goes beyond traditional binary logic. Rather than simply being on or off, each transistor in MLC NAND is able to enter one of four states allowing them to encode data to achieve a storage density of two bits per memory cell, which effectively doubles the capacity of NAND flash memory [16]. In MLC NAND, a page can store 4096 bytes of main data with 128 bytes of spare data. The number of pages in a block is also increased to 128 pages. The current trend for NAND manufacturers is to shift a large percentage of their production to MLC NAND in response to market demands on cost-effective and high-performance NAND flash memory. Table 1 compares the characteristics of the representative large block SLC and MLC NAND chips [20, 21].

NAND flash memory has a restriction that a page should be erased before the new data can be written in the same location. The erase operation can only be performed on a block basis, whose size is larger than a page by 64 or 128 times. In the large block SLC NAND, writing into smaller portions in a page is possible by making use of partial page programming. The NOP in Table 1 refers to the maximum number of partial programming that is allowed for a single page. Note that the NOP in MLC NAND is reduced to one which means that MLC NAND does not support partial page programming. In addition, we can observe from Table 1 that the write operation requires a relatively long latency compared to the read operation. The ratio between write latency to read latency is about 3.3:1 in the large block SLC NAND, and it is increased to 5.5:1 in MLC NAND. Since the write operation sometimes needs to accompany the erase operation, the operational latency becomes even longer.

There are two major approaches to use NAND flash memory as a storage device in embedded systems. One is to employ a software layer called FTL (Flash Translation Layer) between applications and NAND flash memory [1]. The advantage of this approach is that legacy file systems or

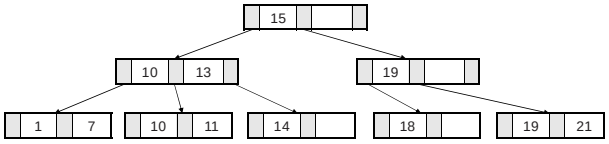


Figure 1: An example of B$^+$-Tree of order 2

DBMSs designed for disks can be used without any modification since FTL emulates the functionality of a normal block device. However, FTL may experience degraded performance if the workload does not show the access pattern for which FTL is optimized. Most FTL algorithms are based on the principle of locality in storage access patterns [10, 11], but the updates to the B$^+$-Tree-like index structure reveal relatively more random access pattern requiring many small flash write operations. In addition, garbage collection, which is the process that reclaims invalid pages by erasing appropriate blocks, may not be efficient because only the limited information is available at block level.

The second approach is to directly manage NAND flash memory without an intermediate layer, as can be seen in flash-aware file systems such as JFFS [24] and YAFFS [2]. These file systems are largely based on the log-structured file system (LFS) [18] which is suitable for NAND flash memory due to the out-of-place update scheme.

2.2 B$^+$-Tree

B$^+$-Tree is a well-known data structure for managing a large amount of data efficiently [6]. B$^+$-Tree keeps data sorted and allows amortized logarithmic-time retrievals, insertions, and deletions. Currently, B$^+$-Tree is used by major DBMSs and file systems [12, 14, 15, 17].

The structure of B$^+$-Tree is similar to that of binary search tree. 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, $K_1, K_2, \cdots, K_d$, and $d+1$ pointers, $P_1, P_2, \cdots, P_{d+1}$. 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 *order* of B$^+$-Tree. Figure 1 illustrates an example B$^+$-Tree of order 2.

There are two types of nodes in B$^+$-Tree: leaf nodes and non-leaf nodes. A leaf node can store from $d/2$ to d keys and the associated pointers. For $i = 1, 2, \cdots, d$, 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. In the case of non-leaf nodes, called *index nodes*, the structure is essentially the same but pointers point to other B$^+$-Tree nodes.

B$^+$-Tree tries to keep its nodes balanced whenever any insertion or deletion occurs. Unbalanced trees such as the 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.

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, however, 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 to 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 [6]. As a result of concatenation, a node can be removed from B$^+$-Tree, and a chain of concatenation may decrease the height of B$^+$-Tree by one in the worst case.

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 Kbytes to 16 Kbytes for modern hard disks considering the seek penalty, the rotational delay, DMA granularity, the unit of paging, and many other factors [8, 22].

2.3 Motivation

The designers of JFFS3, the next version of JFFS2, have adopted B$^+$-Tree for organizing directory entries and locating the positions of the latest file contents. However, because of the absence of FTL, an update to a leaf node leads to a chain of updates on index nodes up to the root node. They call this method of updating tree "*wandering tree*". Any tree may be called wandering tree if an update in the tree requires updating parent nodes up to the root due to the lack of the ability to perform in-place updates [4].

Figure 2 depicts an update example on a wandering tree. The tree is composed of six nodes, A – F. If an update occurs on F, the modified node F' should be written into a new page since flash memory does not support in-place update. The node F', however, is not accessible from the root node because C still points to the obsolete node F. Hence, the node C also needs to be updated and C' is written into flash memory. For the same reason, the new root node A', which points to C', is written.

To reduce the number of updates on index nodes, JFFS3 uses an in-memory data structure called the *journal tree*. When something is changed in the JFFS3 file system, the corresponding leaf node is written into flash memory, but the indexing 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 indexing node updates and lessen the amount of flash

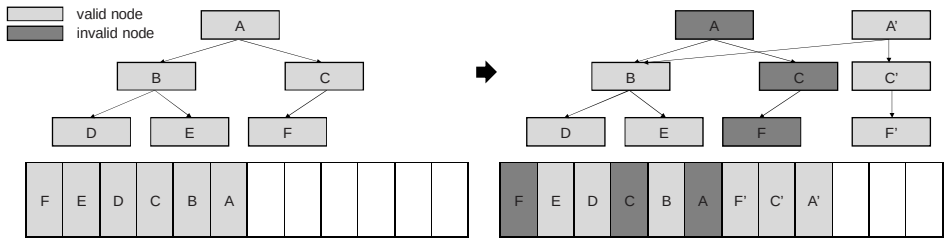


Figure 2: A wandering tree update example

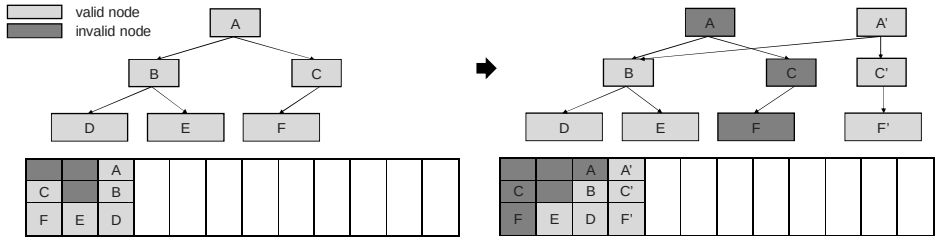


Figure 3: A μ-Tree update example

write operations [4]. 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 [4]. 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

3.1 Overview of μ-Tree

μ-Tree is a balanced tree similar to B$^+$-Tree and is designed not only for SLC NAND but also for MLC NAND. To solve the inefficiency in wandering trees, μ-Tree basically stores all the updated nodes along the path from the root to the leaf into a single flash memory page. Figure 3 illustrates how μ-Tree reduces the number of flash write operations. The number of nodes updated, F', C', and A', is the same as in Figure 2. However, μ-Tree arranges the layout so that those updated nodes fit into a single flash memory page.

We define the *level* of a node in μ-Tree as the length of the path from leaves to the node plus one. The level of a leaf node is always one. Also, the *height* of μ-Tree is defined as the level of the root node. For example, the height of μ-Tree shown in Figure 3 is three.

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 height of μ-Tree. This is because μ-Tree should be able to store all the nodes in a path from the root to any leaf (A', C', and F' in Figure 3) into a single page. Figure 4 depicts the change in the page layout when the page size is 4096 bytes.

Let us denote the height of μ-Tree as H, and the set of nodes in level L as N_L. For μ-Tree such that $H > 2$, a leaf node $n \in N_1$ always occupies the half of the page. As the level is increased, the node size is reduced by half as shown

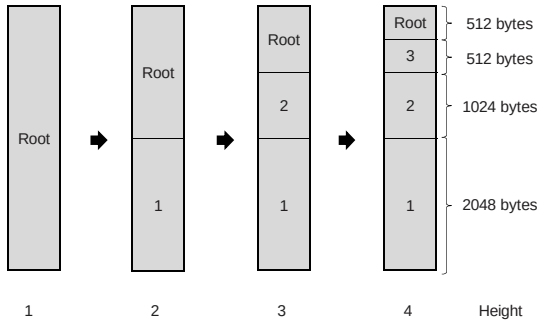


Figure 4: The change in the page layout according to the height of μ-Tree

Algorithm 1 GetNodeFromPage

Input: *page_address* P, *level* L
Output: *node* N
1: $S \Leftarrow Q/2^L$, where Q is the size of a page
2: $O \Leftarrow S$
3: **if** $L = H$ **then**
4: $S \Leftarrow S * 2$
5: $O \Leftarrow 0$
6: **end if**
7: $N \Leftarrow$ read at page P from offset O with size S
8: **return** N

in Figure 3. For an index node m_L, such that $m_L \in N_L, 1 < L < H$, the size of the node is reduced by half compared to its children at the next level $L-1$. Only the root node has the same size as its children nodes. When μ-Tree consists of a single level, i.e., $H = 1$, the entire flash page is used for the root node. The complete procedure to find a node at level L in a page P for μ-Tree with the height H is summarized in Algorithm 1.

The page layout used in μ-Tree gives two benefits. First, when the height of μ-Tree grows or shrinks, we can reuse all the existing nodes (other than the root) since the layout ensures the same node size at each level except for the

Algorithm 2 Retrieval

Input: *key* K (search predicate)
Output: *page_address* O (which points to the record corresponding to K)

```
1: C ⇐ GetNodeFromPage(root page address, H)
2: L ⇐ H
3: while C.type ≠ LEAF do
4:    K_i ⇐ smallest search-key greater than K
5:    L ⇐ L − 1
6:    if K_i exists then
7:       C ⇐ GetNodeFromPage(P_i, L)
8:    else
9:       C ⇐ GetNodeFromPage(P_m, L), where m is the number of pointers in C
10:   end if
11: end while
12: if K_i exists in C, such that K_i = K then
13:    return P_i
14: else
15:    return NULL
16: end if
```

Algorithm 3 Insertion

Input: *key* K, *page_address* P (which points to the record corresponding to K)

```
1: allocate a new page N
2: (R, K′, P′) ⇐ InsertEntry(K, P, N, root page address, H)
3: if R = FULL then
4:    allocate a new page N′
5:    C ⇐ GetNodeFromPage(N, H)
6:    H ⇐ H + 1
7:    (C_l, C_r) = Split(C)
8:    C′ ⇐ GetNodeFromPage(N, H)
9:    insert (C_l.K_1, N) and (C_r.K_1, N′) into C′
10:   write node C_l on page N
11:   write node C_r on page N′
12:   write node C′ on page N′
13: end if
```

root node. If we divide a page evenly among all the levels, it will cause considerable write operations when the height changes because the existing nodes does not fit on the new layout anymore. Second, the layout scheme in μ-Tree devotes at least half of the page to leaf nodes. The size of a leaf node has great influence on the space utilization. More specifically, the larger leaf node size lessens the space overhead. This will be analyzed in more detail in Section 4.3.

Basically, there is no significant difference between B$^+$-Tree and μ-Tree except that the size of node in μ-Tree is determined by its level and the height of the tree. In the following subsections, we describe the distinction between two trees when they handle tree operations such as retrieval, insertion, and deletion.

3.2 Retrieval in μ-Tree

The retrieval process of μ-Tree is essentially the same as that of B$^+$-Tree, because the logical structures of two trees are identical. However, while a page is fully occupied by a node in B$^+$-Tree, a page under μ-Tree may have many nodes at different levels. *GetNodeFromPage*() is used iteratively during the retrieval process to locate the correct node. Algorithm 2 outlines pseudocode for retrieving the record that corresponds to a key K.

3.3 Insertion in μ-Tree

Algorithm 3 shows pseudocode of the insertion process for a key K and the address of the corresponding record P. The *Insertion*() procedure first allocates a new page N to rewrite all the nodes which will be updated. Then, it calls *InsertEntry*() (Algorithm 4) to insert the entry (K, P) into the page N. After finishing the call, it checks the return value. The return value of $FULL$ means that the root node becomes full as a result of the current insertion, and the height of μ-Tree is increased by one.

Similar to B$^+$-Tree, the *InsertEntry*() procedure recursively visits several nodes from the root to the leaf. Normally, it inserts the given entry (K, P) into the leaf node and rewrites all the nodes from the leaf to the root. If the leaf node is already full, however, a new page N' is allocated and a node split occurs. The original node C is divided into C_l and C_r, and they are stored into the page N and N', respectively. The new entry is inserted either into C_l or

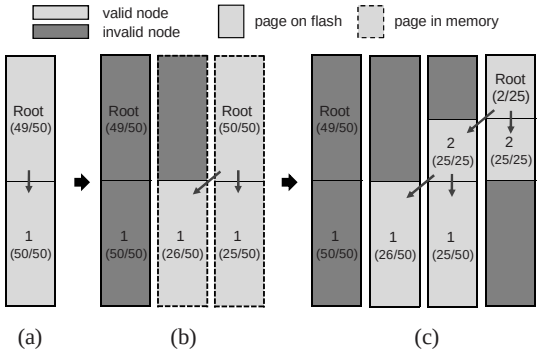


Figure 5: An example of height increase in μ-Tree

into C_r depending on the value of K. Finally, the entry ($C_r.K_1, N'$) is inserted to their parent node. If the parent node becomes full due to this entry, the split process occurs again.

When the root node becomes full, the insertion process increases the height as shown in line 4–12 of Algorithm 3. Figure 5 illustrates an insertion example which causes the height increase. The number in each node indicates the level of the node, and (a/b) represents that the node currently holds a entries out of the maximum b entries. An arrow between two valid nodes means that one has a pointer to the other node.

The initial state of μ-Tree is shown in Figure 5(a) where the leaf node is full and the root node has room for only one more entry. Now assume that an entry is inserted into the leaf node. Calling *InsertEntry*() results in a split on the leaf node and, as a result, an entry is added to the root node. Figure 5(b) shows the intermediate state after returning from *InsertEntry*(). Because the root is full, the root is split into two lower-level nodes increasing the height by one. Figure 5(c) presents the final state of μ-Tree after the insertion operation completes.

Note that in lines 28–30 of Algorithm 4, two nodes C_l and C_r, split from the node C, are swapped each other when some entry in C_r has a pointer to a node in page N. This is because μ-Tree maintains a property that only the direct child node can be stored at the next lower level, if any, within the same page. We call this property the *descendant-ancestor relationship*, which makes it easier to perform garbage collection (cf. Section 5.1).

Algorithm 4 InsertEntry

Input: *key* K, *page_address* P, N, B, *level* L
Output: *return_value* R, *key* K', *page_address* P'

```
1: C ⇐ GetNodeFromPage(B, L)
2: if C.type ≠ LEAF then
3:    find C.P_i, such that C.K_i ≤ K < C.K_{i+1}
4:    if C.P_i doesn't exist then
5:       i ⇐ m, where m is the number of pointers in C
6:    end if
7:    (R, K', P') ⇐ InsertEntry(K, P, N, C.P_i, L − 1)
8:    C.P_i ⇐ N
9:    if R = SPLIT then
10:      K ⇐ K', P ⇐ P', N ⇐ P'
11:   else
12:      write node C on page N
13:      return R ⇐ NULL
14:   end if
15: end if
16: if C has space for (K, P) then
17:    insert (K, P) into C
18:    write node C on page N
19:    if C is full then
20:       return R ⇐ FULL
21:    else
22:       return R ⇐ NULL
23:    end if
24: else
25:    allocate a new page N'
26:    (C_l, C_r) ⇐ Split(C)
27:    insert (K, P) into (C_r.K_1 > K)? C_r : C_l
28:    if C_l.type ≠ LEAF & ∃C_r.P_i = N then
29:       swap C_l ⇔ C_r
30:    end if
31:    write node C_l on page N
32:    write node C_r on page N'
33:    return R ⇐ SPLIT, K' ⇐ C_r.K_1, P' ⇐ N'
34: end if
```

Algorithm 5 Deletion

Input: *key* K

```
1: allocate a new page N
2: R ⇐ DeleteEntry(K, N, root page address, H)
3: if R = ONE then
4:    C ⇐ GetNodeFromPage(N, H)
5:    C' ⇐ GetNodeFromPage(C.P_1, H)
6:    H ⇐ H − 1
7:    C ⇐ GetNodeFromPage(N, H)
8:    C ⇐ C'
9:    write node C' on page N
10: end if
```

3.4 Deletion in μ-Tree

Algorithm 5 shows pseudocode of the deletion process. The *Deletion*() procedure is very similar to *Insertion*() except that the height of the tree may be decreased by one as a result of the deletion.

The *DeleteEntry*() procedure works recursively to delete an entry that matches the given key K. If the leaf node becomes empty by the deletion, the entry in the parent node, which points to the empty node, is also removed. Again, this deletion process can be propagated towards the root node. Due to space limitation, we omit the detailed algorithm for *DeleteEntry*().

μ-Tree shrinks its height when the root node has only one pointer, as shown in lines 4–9 of Algorithm 5. Figure 6 depicts a deletion example which causes the height to decrease. Figure 6(a) shows the initial state of μ-Tree before the deletion is performed. Assume that an entry in the right leaf node in Figure 6(a) is removed. This eventually results in a situation that the root contains only a single pointer as illustrated in Figure 6(b). The final state is shown in Figure 6(c), where the node previously at level 2 is promoted to the root with decreasing the height of the tree.

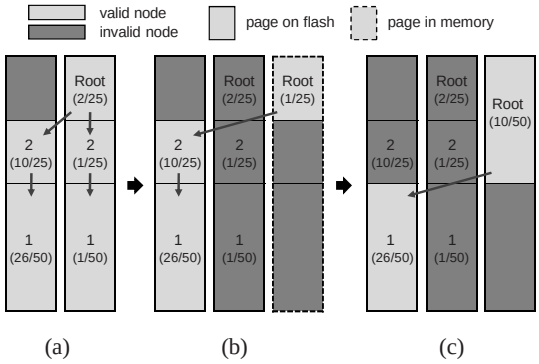


Figure 6: An example of height decrease in μ-Tree

Table 2: Summary of notation

Symbols	Definitions
h_B, h_μ	the height of B$^+$-Tree (h_B) or μ-Tree (h_μ)
l	the level of a node
d_l	the max number of entries within a node at l-th level
f	the max number of entries within a page
n_h	the total number of records to be indexed with a height h tree
v_B, v_μ	the total number of valid pages in B$^+$-Tree (v_B) or μ-Tree (v_μ)
c_r	the cost of read operation on flash memory
c_w	the cost of write operation on flash memory

Our current implementation does not include any optimization for node balancing such as redistribution or concatenation, because it requires additional flash operations. A node simply disappears when it has no entry.

4. SYSTEM ANALYSIS

We develop an analytical model to compare the behavior of μ-Tree and the wandering B$^+$-Tree. For ease of analysis, we assume that the system has no additional memory for cache (or journal tree in B$^+$-Tree), and there are enough free blocks to perform tree operations without any garbage collection. Table 2 summarizes the notation used in our analysis.

4.1 The Cost of Operations

Table 3 shows the cost for each tree operation when there is no node split or removal. Obviously, the cost of a retrieval operation is linearly proportional to the tree height. For insertion and deletion operations, we have to reach a leaf node first and then the leaf node is updated. B$^+$-Tree requires write operations as many as its height, while μ-Tree only a single write operation.

4.2 The Height of Trees

The height of μ-Tree can be slightly higher than that of B$^+$-Tree for the same number of entries. This is because, as

Table 3: The cost of operations

Operations	B$^+$-Tree	μ-Tree
Retrieval	$c_r h_B$	$c_r h_\mu$
Insertion	$(c_r + c_w)h_B$	$c_r h_\mu + c_w$
Deletion	$(c_r + c_w)h_B$	$c_r h_\mu + c_w$

the level increases, the fanout of a node becomes smaller in μ-Tree due to the decreased node size.

First, we analyze the height of B$^+$-Tree in which the node size is same for all nodes. Assuming each node occupies a page, we have

$$d_l = f, \text{ for all } 1 \le l \le h_B \tag{1}$$

We can obtain n_h by multiplying d_l for all levels.

$$n_h = \prod_{i=1}^{h} d_i = f^h \tag{2}$$

From (2), h_B is expressed as follows for n records.

$$h_B = \log_f n \tag{3}$$

In μ-Tree, d_l is given by

$$d_l = \begin{cases} f/2^{l-1} & \text{if root } (l = h_\mu); \\ f/2^l & \text{otherwise } (l < h_\mu). \end{cases} \tag{4}$$

In the same manner,

$$n_h = \prod_{i=1}^{h} d_i = 2\prod_{i=1}^{h}(f/2^i) \tag{5}$$

From (5), h_μ for n records can be represented as follows.

$$h_\mu = -\log_2 \frac{\sqrt{2}}{f} - \sqrt{\log_2^2 \frac{\sqrt{2}}{f} - 2\log_2 \frac{n}{2}} \tag{6}$$

Figure 7 plots h_B, $\lceil h_B \rceil$, h_μ, and $\lceil h_\mu \rceil$ obtained from (3) and (6) when the page size is 4 Kbytes and the entry size is 8 bytes (i.e., $f = 512$). We can see that the height of μ-Tree is equal to or at most one level higher than that of B$^+$-Tree up to one billion records. Although the increased tree height raises the number of flash read operations during retrieval, insertion, and deletion, the impact on the overall performance can be alleviated by the fact that the read operation is much faster than the write operation on NAND flash memory.

4.3 Space Overhead

As mentioned in Section 3, μ-Tree requires more space than B$^+$-Tree for the same number of records because the fanout of the leaf node is smaller. Assuming that every leaf node is filled with entries, v_B can be calculated as follows.

$$v_B = \left\lceil \frac{n}{d_1} \right\rceil + \left\lceil \frac{n}{d_1 d_2} \right\rceil + \ldots + \left\lceil \frac{n}{d_1 d_2 \ldots d_{h_B}} \right\rceil \tag{7}$$

By applying (1),

$$v_B = \left\lceil \frac{n}{f} \right\rceil + \left\lceil \frac{n}{f^2} \right\rceil + \ldots + \left\lceil \frac{n}{f^{h_B}} \right\rceil \tag{8}$$

In most cases f is sufficiently large ($f > 100$), and we can ignore $\left\lceil \frac{n}{f^2} \right\rceil + \ldots + \left\lceil \frac{n}{f^{h_B}} \right\rceil$. Hence v_B is given by

$$v_B \approx \left\lceil \frac{n}{f} \right\rceil \tag{9}$$

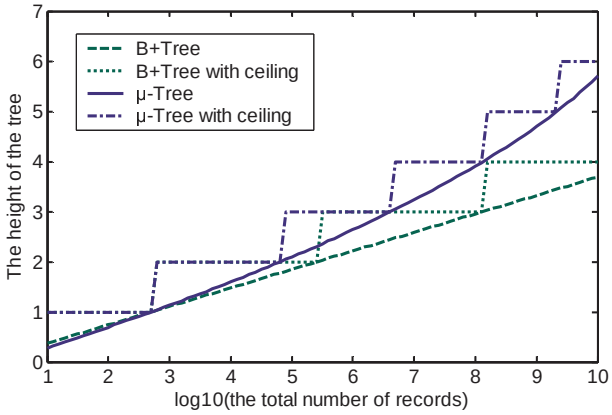


Figure 7: Comparison of the height of B$^+$-Tree and μ-Tree

In μ-Tree, we can only count leaf nodes because index nodes are stored with leaf nodes. Therefore, v_μ can be expressed as

$$v_\mu = \left\lceil \frac{n}{d_1} \right\rceil = \left\lceil \frac{n}{f/2} \right\rceil \tag{10}$$

From (9) and (10) we have

$$v_B \approx \frac{v_\mu}{2} \tag{11}$$

μ-Tree takes up approximately twice as many as flash memory pages compared to B$^+$-Tree. The space overhead of μ-Tree can be reduced by allocating more space to leaf nodes inside a page. Often the number of valid pages is not directly related to the overall space utilization on a system using the out-of-place update scheme since nodes become invalid as the tree is updated. Note that B$^+$-Tree generates more invalid pages than μ-Tree during insertion or deletion operations (cf. Section 6.3).

5. IMPLEMENTATION ISSUES

5.1 Garbage Collection

When the number of free blocks goes down below a certain threshold, the system needs to trigger garbage collection to reclaim invalidated flash memory pages. Once the garbage collection process is triggered, we first have to choose a victim block which will be cleaned. The ideal candidate that minimizes the cost of garbage collection is the block which has the minimum number of valid pages. Unfortunately, keeping track of the number of valid pages for all blocks limits the scalability of μ-Tree. Instead, we maintain the information for a small number of blocks whose pages are invalidated after the system boots.

If the victim block is selected, the next step is to separate valid pages from invalid pages. A page is invalid if all the nodes in the page are not reachable from the root node. In order to minimize the time and the space needed to identify valid pages, we store the information on the validity of every page into the spare area of the last page in each block. Since a page can be invalidated anytime after the validity information is written into flash memory, the garbage collection process checks again whether the page marked as valid is still reachable from the root node. The descendant-ancestor

Table 4: The characteristics of traces used for evaluation

Trace	Description	Retrieval	Insertion	Deletion
kernel_compile	This trace is obtained while we untar, compile, clean, and remove the Linux kernel source.	2,274,867	260,974	236,027
postmark	This is a trace for postmark benchmark. This benchmark models the workload of an e-mail server, where lots of files are created and deleted.	4,617,494	574,148	391,847
mp3	This is a trace for synthetic workload, where we model the usage scenario of a MP3 player. The storage of 8 Gbytes is filled with MP3 files fully, and then 50% of files are deleted and copied again five times.	512,986	223,188	23,950

relationship described in Section 3.3 makes it easier to perform this test; it is enough to test only the reachability of the node at the lowest level in a page to see if the page is valid or not. According to the descendant-ancestor relationship, if the node at the lowest level is invalid, no valid parent nodes can exist in the same page. During garbage collection, the valid page can be moved to another block simply by updating any key in the lowest-level node with the same value.

5.2 Recovery

We also need a resilient recovery mechanism that can cope with sudden power failure. μ-Tree basically relies on checkpointing, where the information including the tree height, the blocks used by μ-Tree, and the address of the root node is periodically written into the special checkpoint area in NAND flash memory. In our current implementation, the checkpoint area is organized similar to the superblock management scheme of JFFS3 [4], and the block containing the checkpoint information is indirectly referenced from the fixed location.

During initialization, the system first checks whether it is turned off normally. If the system has terminated abnormally, the recent checkpoint information is loaded and the location of the root node is identified by following the pages updated after the last checkpoint operation. Even when the crash occurs during an insertion or a deletion which involves more than one flash write operations, we can easily recover from the situation because μ-Tree ensures that the page containing the root node is written in the end. Pages that are not ended with the valid root node can be simply discarded.

5.3 Caching

The use of in-memory cache such as the JFFS3's journal tree [4] is effective in reducing the number of read and write operations on flash memory. By keeping the nodes accessed by previous operations in memory, subsequent read requests can be satisfied without any flash operation. Likewise, the updated nodes are temporarily buffered in the cache which can absorb subsequent write requests to the same node.

We have implemented a cache system which consists of separate read and write cache. Both caches are maintained on a page basis and the contents are mutually exclusive. The pages in the read cache are managed by an LRU list, while those in the write cache are handled in an allocation order.

For read requests from μ-Tree, our cache system first examines the read cache, and then the write cache. If the requested page does not exist in both caches, a read command is sent to the flash device. When μ-Tree writes a new page, a page allocation request is issued to the write cache. If the write cache is full, all the pages in the write cache are written in bulk into the flash device. For insertion operation which generates more than one pages due to node split, the cache flush is delayed until the end of each operation in order to guarantee the atomicity of operation. A page can become invalid while it is in the cache, in which case the page slot in the cache is immediately reused.

6. EXPERIMENTAL EVALUATION

6.1 Evaluation Methodology

μ-Tree has been built on a NAND flash simulator which has an ability to count the number of flash read, write, and erase operations performed. For a fair comparison, we have also implemented the wandering B^+-Tree scheme proposed by JFFS3 [4] on the same simulator. The simulator is configured for 64MB MLC NAND flash memory with 4 Kbytes page size and 512 Kbytes block size. It is used exclusively for storing indexes (excluding actual records).

The performance of μ-Tree is investigated by microbenchmark and traces extracted from real workloads. We have developed our own microbenchmark where, after inserting one million records initially, we conduct ten thousands of retrievals followed by the same number of deletions and then insertions with randomly generated key values. The microbenchmark is used to compare the average number of flash memory operations required for each retrieval, insertion, and deletion operation.

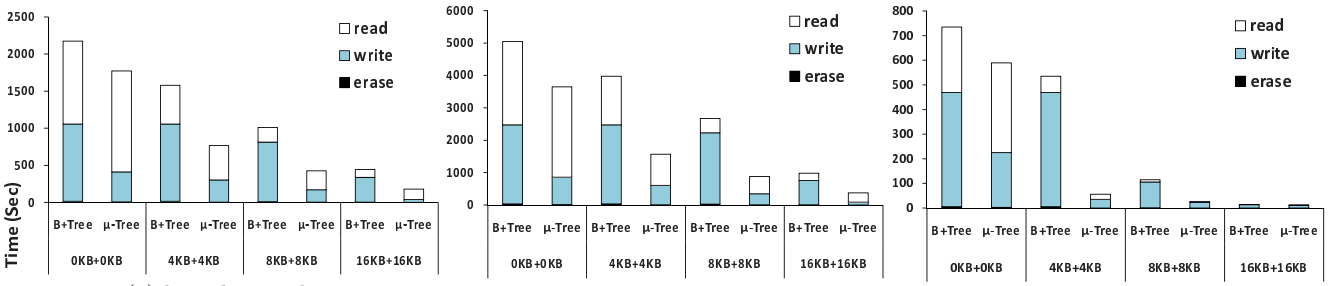
To evaluate the performance of μ-Tree for real workloads, we collected traces of B^+-Tree operations from Reiser file system (ReiserFS) [17]. ReiserFS is one of the representative disk-based file systems in Linux and it uses B^+-Tree extensively for indexing directory entries and the metadata of files. We have instrumented the internal of ReiserFS on Linux kernel 2.6.16.1 to log every B^+-Tree operation while we execute several tasks which require lots of metadata operations. Table 4 summarizes the characteristics of traces used in our experiments.

6.2 Microbenchmark Results

Table 5 summarizes the average number of flash read and write operations required for each retrieval, insertion, and deletion operation in B^+-Tree and μ-Tree. The cache size is represented as a pair of the read cache size and the write cache size. The first configuration denoted as "0KB+0KB" is the case where the cache is not used at all. The flash memory access cost is calculated based on the access latencies of MLC NAND shown in Table 1. Since garbage collection is invoked during the execution of microbenchmark, the flash erase operation is also performed once in a while. In table 5, we omit the number of flash erase operations since the number is very small in most cases (< 0.03 in B^+-Tree

Table 5: Microbenchmark results

Cache size (Read+Write)	Metrics	Retrieval		Insertion		Deletion	
		B^+-Tree	μ-Tree	B^+-Tree	μ-Tree	B^+-Tree	μ-Tree
0KB+0KB	Average number of flash read / operation	3.00	2.97	4.11	3.32	4.14	3.34
	Average number of flash write / operation	0.00	0.00	3.83	1.08	3.85	1.09
	Flash memory access cost (ms)	0.50	0.50	4.19	1.55	4.19	1.54
4KB+4KB	Average number of flash read / operation	2.00	1.97	2.83	2.74	2.83	2.76
	Average number flash write / operation	0.00	0.00	3.83	1.08	3.83	1.09
	Flash memory access cost (ms)	0.33	0.33	3.98	1.45	3.98	1.45
8KB+8KB	Average number of flash read / operation	1.77	1.94	2.40	2.49	2.40	2.51
	Average number of flash write / operation	0.00	0.00	3.84	1.08	3.85	1.09
	Flash memory access cost (ms)	0.29	0.32	3.92	1.40	3.93	1.41
16KB+16KB	Average number of flash read / operation	1.67	1.90	2.03	2.29	1.82	2.08
	Average number of flash write / operation	0.00	0.00	2.75	1.08	2.74	1.09
	Flash memory access cost (ms)	0.28	0.31	2.85	1.37	2.84	1.38



(a) kernel_compile (b) postmark (c) mp3

Figure 8: The total elapsed time to replay tree operations for each trace

and < 0.01 in μ-Tree). However, the erase cost is reflected in computing the flash memory access cost.

From Table 5, we can observe that both trees benefit from the increased cache size. When there is no cache, each retrieval operation requires almost three flash read operations. This is obvious because the height of B^+-Tree or μ-Tree grows to three for one million records. When the cache can hold two pages, one for the read cache and another for the write cache, the costs for retrieval lessen in both trees because the root node is kept in write cache. The read cache is not useful for B^+-Tree since B^+-Tree needs to start from the root node every time discarding the leaf node previously read. In μ-Tree, however, there can be a cache hit in the same condition since a page holds several nodes in different levels. For the read cache size larger than 8 Kbytes, μ-Tree requires slightly more number of flash read operation compared to B^+-Tree. This is because currently the cache is managed on a page basis, and μ-Tree does not utilize the entire cache area effectively due to the unused area and invalid nodes inside a page.

For insertion or deletion operations, μ-Tree outperforms B^+-Tree by around three times in flash memory access cost regardless of the cache configuration. Note that when there is no cache, the values in Table 5 are higher than the values predicted by Table 3 due to garbage collection.

6.3 Real Workloads Results

Figure 8 compares the time to replay tree operations in B^+-Tree and μ-Tree for traces shown in Table 4. We breakdown the total elapsed time according to the time spent performing flash read, write, and erase operations.

When there is no cache, μ-Tree improves the total execution time by 18%, 28%, and 20% for kernel_compile, postmark, and mp3 traces, respectively, compared to B^+-Tree. Although μ-Tree requires slightly more number of flash read operations, most speedup comes from the significant reduction in the number of flash write operations. μ-Tree requires only one third of flash write operations for kernel_compile and postmark, and half for mp3. Again, the improvement is related to the height of B^+-Tree, which grows to three for kernel_compile and postmark, and to two for mp3.

In accordance with microbenchmark results, the cache is useful for both B^+-Tree and μ-Tree, but we can see that it is more effective to μ-Tree. With a small cache of 8 Kbytes (4KB+4KB), μ-Tree exhibits the performance improved by 51%, 61%, and 90% for kernel_compile, postmark, and mp3, respectively, compared to B^+-Tree with the same cache size. The notable speedup in mp3 is due to sequential insertion operations to index the file allocation information in ReiserFS. Since the write cache can hold just a single page with 4 Kbytes, B^+-Tree cannot fully utilize the write cache during the repeated insertion operations.

Figure 9 illustrates the number of garbage collection invoked during the simulation of each trace when there is no cache. Results for other cache configurations are also very similar. Although μ-Tree has twice as many valid pages as B^+-Tree (cf. Section 4.3), garbage collection is invoked more frequently for B^+-Tree. This is because B^+-Tree produces more number of invalid pages during insertion or deletion, suffering from the shortage of free blocks.

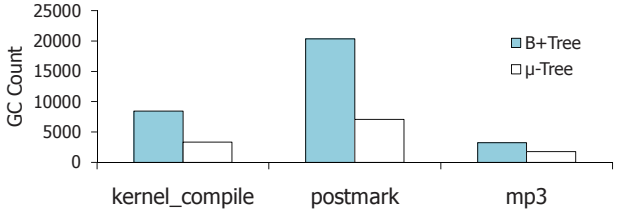


Figure 9: The total number of garbage collections invoked during the simulation with no cache

7. RELATED WORK

Recently, several studies have been performed to develop an efficient index structure for flash memory. Wu et al. introduced BFTL, an optimized B^+-Tree layer for flash memory [5]. BFTL does not suffer from the out-of-place update problem of flash media because it was implemented on top of FTL. They mainly focused on the trade-off between retrieval and update performance. To avoid expensive update cost for each node, all changes are written in log pages. For each node, an in-memory data structure called *Node Translation Table (NTT)* maintains a list of log pages that have inserted or removed entries. While BFTL reduces the write cost, the read cost is degraded by a factor of two or more because many log pages related to the requested node must be collected.

To address this problem, Nath et al. proposed a hybrid approach [13]. They classify nodes into read-intensive and write-intensive nodes by their cost model. To avoid the read overhead for the read-intensive nodes, those nodes are disallowed to use the log pages, thus they are written entirely on updates. Those two approaches, however, can not be adopted on small embedded systems with limited memory because the memory footprint of NTT is proportional to the number of nodes. Moreover, both approaches incur additional cost because they are based on FTL. The updates to B^+-Tree reveal many small writes which are distributed randomly across many blocks, but most FTL algorithms perform poorly for such a workload [1, 11]. The use of log pages does not lessen the problem since the log entries in the same node are eventually merged together.

Lin et al. proposed MicroHash [19], an indexing structure based on hash table for sensor devices. However, MicroHash is not scalable to a large number of records. It is also hardly applicable to file systems or DBMSs because it assumes that all stale records written before a certain amount of time are abandoned in the deletion phase.

8. CONCLUSIONS AND FUTURE WORK

This paper proposes a novel index structure called μ-Tree that can handle tree update requests efficiently on NAND flash memory by storing all the nodes in the path from the root to the leaf into a single flash page. Our evaluation results show that μ-Tree outperforms B^+-Tree by up to 28% for traces extracted from real workloads. When the system is able to utilize a small in-memory cache of 8 Kbytes, μ-Tree improves the overall performance by up to 90% compared to B^+-Tree with the same cache size.

As mentioned in Section 4.3, the space used by the root node need not necessarily be partitioned equally when the height of μ-Tree grows. In fact, allocating more space to the node in the lower level increases the overall space utilization at the expense of the increase height. Our future work includes the complete analysis on the impact of this trade-off. We also plan to investigate more effective cache management policies for μ-Tree.

9. REFERENCES

[1] A. Kawaguchi *et al.* A Flash-Memory Based File System. In *Proc. of the 1995 Winter USENIX Conference*, pages 155–164, 1995.

[2] Aleph One Limited. *Yet Another Flash File System (YAFFS)*. http://www.aleph1.co.uk/yaffs.

[3] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Infomatica*, 1(3):173–189, 1972.

[4] A. B. Bityutskiy. *JFFS3 design issues*. http://www.linux-mtd.infradead.org.

[5] C. H. Wu *et al.* An Efficient B-Tree Layer for Flash Memory Storage Systems. In *Proc. of Int. Conf. on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, 2003.

[6] D. Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[7] E. Gal and S. Toledo. A Transactional Flash File System for Microcontrollers. In *Proc. of the USENIX Annual Technical Conference*, 2005.

[8] J. Gray and G. Graefe. The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *ACM SIGMOD Record*, 26(4):63–68, 1997.

[9] G.-J. Kim *et al.* LGeDBMS: A Small DBMS for Embedded System with Flash Memory. In *Proc. of the 32nd Int. Conf. on Very Large Data Bases (VLDB)*, pages 1255–1258, 2006.

[10] J. Kim *et al.* A Space Efficient Flash Translation Layer for CompactFlash Systems. *IEEE Trans. Consumer Electronics*, 48(2):366–375, May 2002.

[11] J.-U. Kang *et al.* A Superblock-based Flash Translation Layer for NAND Flash Memory. In *Proc. of Int. Conf. on Embedded Systems Software (EMSOFT)*, 2006.

[12] Microsoft Corp. *SQL Server 2005*. http://www.microsoft.com/sql/default.mspx.

[13] S. Nath and A. Kansal. FlashDB : Dynamic Self-Tuning Database for NAND Flash. In *Proc. of Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2007.

[14] Oracle Corp. *Oracle 10g*. http://www.oracle.com/database/index.html.

[15] D. Philips. A Directory Index for Ext2. In *Proc. of the 2001 Annual Linux Showcase and Conference*, 2001.

[16] R. Quinnell. Multi-Level Cell NAND Flash. *http://www.edn.com/article-partner/CA503389.html*, Feb. 2005.

[17] H. Reiser. *Reiser File System*. http://www.namesys.com.

[18] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of A Log-Structured File System. *ACM Trans. Computer Systems*, 10(1):26–52, 1992.

[19] S. Lin *et al.* Efficient Indexing Data Structures for Flash-Based Sensor Devices. *ACM Trans. Storage*, 2(4):468–503, November 2006.

[20] Samsung Elec. *2Gx8 Bit NAND Flash Memory (K9GAG08U0M-P)*. 2006.

[21] Samsung Elec. *2Gx8 Bit NAND Flash Memory (K9WAG08U1A)*. 2006.

[22] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concept*. 4th edition, 2002.

[23] M. Slocombe. Samsung CEO: NAND Flash Will Replace Hard Drives. *http://digital-lifestyles.info/display_page.asp?section=platforms&id=2573*, Sep. 2005.

[24] D. Woodhouse. JFFS: The Journalling Flash File System. In *Proc. of the Ottawa Linux Symposium*, 2001.