An Efficient B-Tree Layer for Flash-Memory Storage Systems

Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo
{d90003,d6526009,ktw}@csie.ntu.edu.tw
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan, 106
Fax: +886-2-23628167

Abstract

With a significant growth of the markets for consumer electronics and various embedded systems, flash memory is now an economic solution for storage systems design. For index structures which require intensively fine-grained updates/modifications, block-oriented access over flash memory could introduce a significant number of redundant writes. It might not only severely degrade the overall performance but also damage the reliability of flash memory. In this paper, we propose a very different approach which could efficiently handle fine-grained updates/modifications caused by B-Tree index access over flash memory. The implementation is done directly over the flash translation layer (FTL) such that no modifications to existing application systems are needed. We demonstrate that the proposed methodology could significantly improve the system performance and, at the same time, reduce the overheads of flash-memory management and the energy dissipation, when index structures are adopted over flash memory.

Keywords: Flash Memory, B-Tree, Storage Systems, Embedded Systems, Database Systems.

1 Introduction

Flash memory is a popular alternative for the design of storage systems because of its shock-resistant, power-economic, and non-volatile nature. In recent years, flash-memory technology advances with the wave of consumer electronics and embedded systems. There are significant technology breakthroughs in both of its capacity and reliability features. The ratio of cost and capacity has being increased dramatically. Flash-memory storage devices of 1GB will soon be in the market. Flash memory could be considered as an alternative to replace hard disks in many applications. The implementation of index structures, which are very popular in the organization of data over disks, must be now considered over flash memory. However, with the very distinct characteristics of flash memory, traditional designs of index structures could result in a severe performance degradation to a flash-memory storage system and significantly reduce the reliability of flash memory.

There are two major approaches in the implementations of flash-memory storage systems: The native file-system approach and the block-device emulation approach. For the native file-system approach, JFFS/JFFS2[5], LFM[13], and YAFFS [2] were proposed to directly manage raw flash memory. The file-systems under this approach are very similar to the log-structured file-systems (LFS) [18]. This approach is natural for the manipulation of flash memory because the characteristics of flash memory do not allow in-place updates (overwriting). One major advantage of the native file-system approach is robustness because all updates are appended, instead of

overwriting existing data (similar to LFS). The block-device emulation approach is proposed for a quick deployment of flash-memory technology. Any well-supported and widely used (disk) file-systems could be built over a flash memory emulated block-device easily. For example, FTL/FTL-Lite [10, 11, 12], CompactFlash [4], and SmartMedia [23] are popular block device emulation, which provide a transparent block-device emulation. Regardless of which approach is adopted, they share the similar technical issues: How to properly manage garbage collection and wear-leveling activities.

With the increasing popularity of flash memory for storage systems (and the rapid growing of the capacity), the implementations of index structures could become a bottleneck on the performance of flash-memory storage systems. Hash tables and search trees are popular data structures for data organization. In particular, B-Tree is one of the most popular index structures because of its scalability and efficiency. B-Tree indices were first introduced by Bayer and McCreight [22]. Comer [6] later proposed its variation called B+-tree indices in 1979. B-Tree index structures are extended to many application domains: Kuo, et al. [24] demonstrated how to provide a predictable performance with B-Tree. Freeston [20] showed multi-dimensiona B-Trees which have good predictable and controllable worst-case characteristics. For the parallel environment, Yokota, et al. proposed Fat-Btrees [8] to improve high-speed access for parallel database systems. Becker, et al. [3] improved the availability of data by a multi-version index structure that supports insertions, deletions, range queries, and exact match queries for the current or some past versions.

There are two critical issues which could have a significant impacts on the efficiency of index structures over flash memory: (1) write-once with bulk-erase (2) the endurance issue. Flash memory could not be over-written (updated) unless it is erased. As a result, out-of-date (or invalid) versions and the latest copy of data might co-exist over flash memory simultaneously. Furthermore, an erasable unit of a typical flash memory is relatively large. Valid data might be involved in the erasing, because of the recycling of available space. Frequent erasing of some particular locations of flash memory could quickly deteriorate the overall lifetime of flash memory (the endurance issue), because each erasable unit has a limited cycle count on the erase operation.

In this paper, we focus on an efficient integration of B-Tree index structures and the block-device emulation mechanism provided by FTL (flash translation layer). We propose a module over a traditional FTL to handle intensive byte-wise operations due to B-tree access. The implementation is done directly over FTL such that no modifications to existing application systems are needed. The intensive byte-wise operations are caused by record inserting, record deleting, and B-tree reorganizing. For example, the insertion of a record in the system will result in the insertion of a data pointer at a leaf node and, possibly, the insertion of tree pointers in the B-tree. Such actions could result in a large number of data copyings (i.e., the copying of unchanged data and tree pointers in related nodes) because of out-place updates over flash memory. We demonstrate that the proposed methodology could significantly improve the system performance and, at the same time, reduce the overheads of flash-memory management and the energy dissipation, when index structures are adopted over flash memory. We must point that although only the block-device emulation approach is studied in this paper, however, the idea of this paper could be easily extended to a native flash-memory file system.

The rest of this paper is organized as follows: Section 2 provides an overview of flash memory and discussions of the implementation problems of B-Tree over flash memory. Section 3 introduces our approach and its implementation. Section 4 provides performance analysis of the approach. Section 5 shows experimental results. Section 6 is the conclusion and future work.

2 Motivation

In this section, we shall briefly introduce the characteristics of flash memory. By showing the very distinct properties of flash memory, the potential issues of building a B-Trees index structure over a NAND flash memory

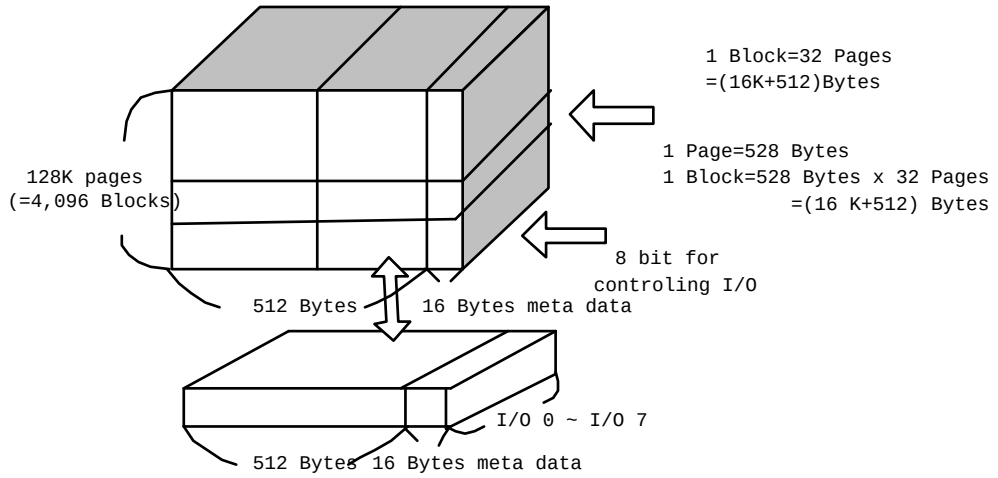


Figure 1: The Flash Memory Organization.

are addressed as the motivation of this work.

2.1 Flash Memory Characteristics

Figure 1 shows the organization of a typical NAND[1] flash memory. A NAND flash memory is organized by many blocks, and each block is of a fixed number of pages. A block is the smallest unit of erase operation, while reads and writes are handled by pages. The typical block size and page size of a NAND flash memory is 16KB and 512B, respectively. Because flash memory is write-once, we do not overwrite data on update. Instead, data are written to free space, and the old versions of data are invalidated (or considered as dead). The update strategy is called "out-place update". In other words, any existing data on flash memory could not be over-written (updated) unless it is erased. The pages store live data and dead data are called "live pages" and "dead pages", respectively. Because out-place update is adopted, we need a dynamic address translation mechanism to map a given LBA (logical block address) to the physical address where the valid data reside. Note that a "logical block" usually denotes a disk sector. To accomplish this objective, a RAM-resident translation table is adopted. The translation table is indexed by LBA's, and each entry of the table contains the physical address of the corresponding LBA. If the system reboots, the translation table could be re-built by scanning the flash memory. Figure 2 illustrate how to retrieve data from flash memory by using the translation table.

After a certain number of page writes, free space on flash memory would be low. Activities consist of a series of read/write/erase with the intention to reclaim free spaces would then start. The activities are called "garbage collection", which is considered as overheads in flash-memory management. The objective of garbage collection is to recycle the dead pages scattered over the blocks so that they could become free pages after erasings. How to smartly choose which blocks should be erased is the responsibility of a *block-recycling policy*. The block-recycling policy should try to minimize the overhead of garbage collection (caused by live data copyings). Figure[2] 3 shows the procedure of garbage collection. Under current technology, a flash-memory block has a limitation on

[1]There are two major types of flash memory in the current market: NAND flash and NOR flash. The NAND flash memory is specially designed for data storage, and the NOR flash is for EEPROM replacement.

[2]A similar figure also appears in [19].

the erase cycle count. For example, a block of a typical NAND flash memory could be erased for 1 million (10^6) times. After that, a worn-out block could suffer from frequent write errors. A "wear-leveling" policy intends to erase all blocks on flash memory evenly, so that a longer overall lifetime could be achieved. Obviously, wear-leveling activities would impose significant overheads to the flash-memory storage system if the access patterns try to frequently update some specific data.

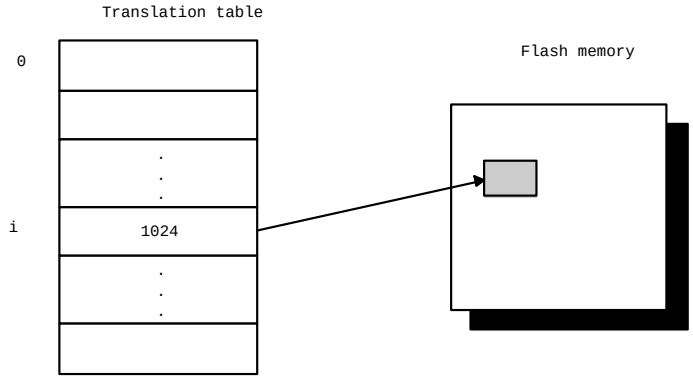


Figure 2: The logical block address "i" is mapped to the physical page number "1024" by the translation table.

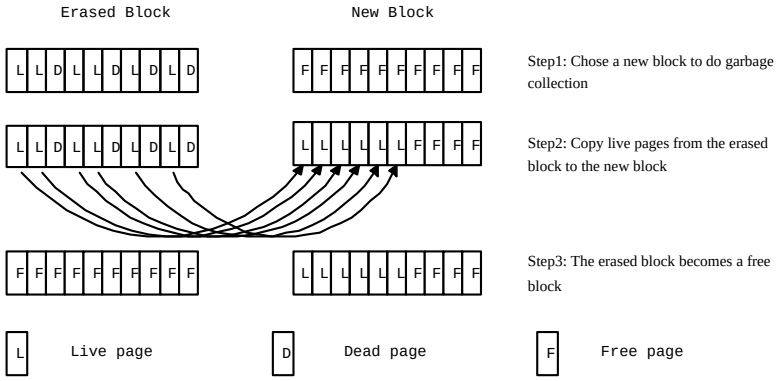


Figure 3: Garbage collection

There are many issues in the management of flash memory: As mentioned in the previous two paragraphs, the activities of garbage collection and wear-leveling could introduce an unpredictable blocking time to time-critical applications. In particular, Kawaguchi, et al. [1] proposed the cost-benefit policy which uses a value-driven heuristic function as a block-recycling policy. Kwoun, et al. [14] proposed to periodically move live data among blocks so that blocks have more an even life-time. Chang and Kuo [15] investigated how to properly manage the internal activities so that a deterministic performance could be provided. On the other hand, the performance and energy consumption of reads, writes, and erases are very different, as shown in Table 1. For portable devices, the endurance of batteries is a critical issue. Because flash memory could also contribute a significant portion of energy consumption, Chang and Kuo [16] introduced an energy-efficient request scheduling algorithm for flash-memory storage system to lengthen the operating time of battery-powered portable devices. Furthermore, the handling of writes could be the performance bottleneck: Writing to flash memory are relatively slow, and it could introduce garbage collection and wear-leveling activities. To improve the overall performance,

	Page Read 512 bytes	Page Write 512 bytes	Block Erase 16K bytes
Performance(μs)	348	909	1,881
Energy Consumption($\mu joule$)	99	237.6	422.4

Table 1: Performance of a typical NAND Flash Memory

Chang and Kuo [17] proposed an adaptive striping architecture which consists of several independent banks. A dynamic striping policy was adopted to smartly distribute writes among banks to improve the parallelism.

2.2 Problem Definition

A B-Tree consists of a hierarchical structure of data. It provides efficient operations to find, delete, insert, and traverse the data. There are two kinds of nodes in a B-Tree: internal nodes and leaf nodes. A B-Tree internal node consists of a ordered list of key values and linkage pointers, where data in a subtree have key values between the ranges defined by the corresponding key values. A B-Tree leaf node consists of pairs of a key value and its corresponding record pointer. In most cases, B-Trees are used as external (outside of RAM) index structures to maintain a very large set of data. Traditionally, the external storage are usually block devices such as disks. In practice, we usually set the size of a B-Tree node as the size which can be efficiently handled by the used block device. For example, many modern hard disks could have equivalent response times to access a 512B sector and a 64KB chunk (due to the seek penalty, the rotational delay, the DMA granularity, and many other factors). Therefore, a B-Tree node could be a 64K chunk on the hard disk. To insert, delete, and re-balance B-Trees, B-Tree nodes are fetched from the hard disk and then written back to the original location. Such operations are very efficient for hard disks.

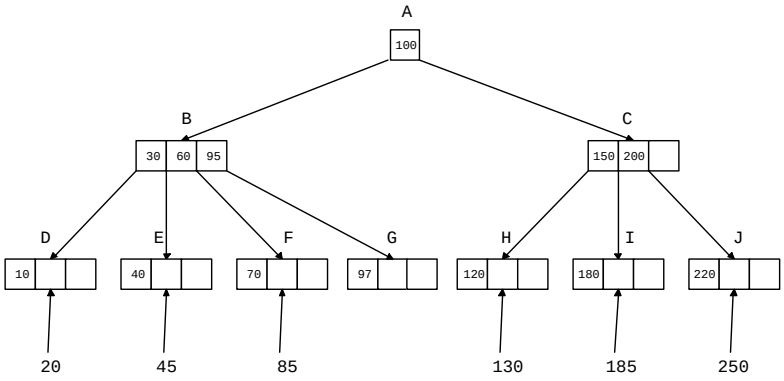


Figure 4: A B-Tree (fanout is 4).

Recently, the capacity and reliability of flash memory grew significantly. Flash-memory storage systems become good mass storage solutions, especially for those applications work under extreme environments. For example, those systems operate under severe vibrations or limited energy sources might prefer flash-memory storage systems. Since a large flash-memory storage system is much more affordable than ever, the issue on the efficiency of data accessing becomes critical. For the development of many information systems, B-Tree are widely used because of its efficiency and scalability. However, a direct adoption of B-Tree index structures over flash-memory storage systems could exaggerate the overheads of flash-memory management. Let us first consider

usual operations done over B-Tree index structures: Figure 4 shows an ordinary B-Tree. Suppose that six different records are to be inserted. Let the primary keys of the records be 20, 45, 85, 130, 185, and 250, respectively. As shown in Figure 4, the 1st, 2nd, 3rd, 4th, 5th, and 6th records should be inserted to nodes D, E, F, H, I, and J, respectively. Six B-Tree nodes are modified. Now let us focus on the files of index structures since we usually store index structures separately from the records. Suppose that each B-Tree node is stored in one page, then up to six page writes are needed to accomplish the updates. If rebalancing is needed, more updates of internal nodes will be needed.

Compared with operations on hard disks, updating (or writing) data over flash memory is a very complicated and expensive operation. Since out-place update is adopted, a whole page (512B) which contains the new version of data will be written to flash memory, and previous data must be invalidated. The page-based write operations could expectedly introduce a sequence of negative effects. Free space on flash memory could be consumed very quickly. As a result, garbage collection could happen frequently to reclaim free space. Furthermore, because flash memory is frequently erased, the lifetime of the flash memory would be reduced. Another problem is energy consumption. Out-place updates would result in garbage collection, which must read and write pages and erase blocks. Because writes and erases consume much more energy than reads, as shown in Table 1, out-place updates eventually cause much more energy consumption. For portable devices, because the amount of energy provided by batteries is limited, energy-saving could be a major concern. The motivation of this work is to reduce the amount of redundant data written to flash memory caused by index structures to improve the system performance and reduce energy consumption.

3 The Design and Implementation of BFTL

In this section, we present an efficient B-Tree layer for flash-memory storage systems (BFTL) with a major objective to reduce the redundant data written due to the hardware restriction of a NAND flash memory. We shall illustrate the architecture of a system which adopts BFTL and present the functionalities of the components inside BFTL in the following subsections.

3.1 Overview

In our approach, we propose to have an insertable module called BFTL (an efficient B-Tree layer for flash-memory storage systems, referred as BFTL for the rest of this paper.) over the original flash translation layer (FTL). BFTL sits between the application layer and the block-device emulated by FTL. The BFTL module is dedicated to those applications which use services provided by B-Tree indices. Figure 5 illustrates the architecture of a system which adopts BFTL. BFTL consists of a small *reservation buffer* and a *node translation table*. B-Tree index services requested by the upper-level applications are handled and translated by BFTL, and then block-device requests are sent from BFTL to FTL. When the applications insert, delete, or modify records, the newly generated records (referred as "dirty records" for the rest of this paper) would be temporarily held by the reservation buffer of BFTL. Since the reservation buffer only holds an adequate amount of records, the dirty records should be timely flushed to flash memory. Note that record deletions are handled by adding "invalidation records" to the reservation buffer.

To flush out the dirty records in the reservation buffer, BFTL constructs corresponding "index units" for each dirty record. The usage of index units are to reflect primary-key insertions and deletions to the B-Tree index structure caused by the dirty records. The storing of the index units and the dirty records are handled in two different ways. The storing of the records is relatively simple: The records are written (or updated) to an allocated (or the original) locations. On the other hand, because an index unit is very small (compared with the

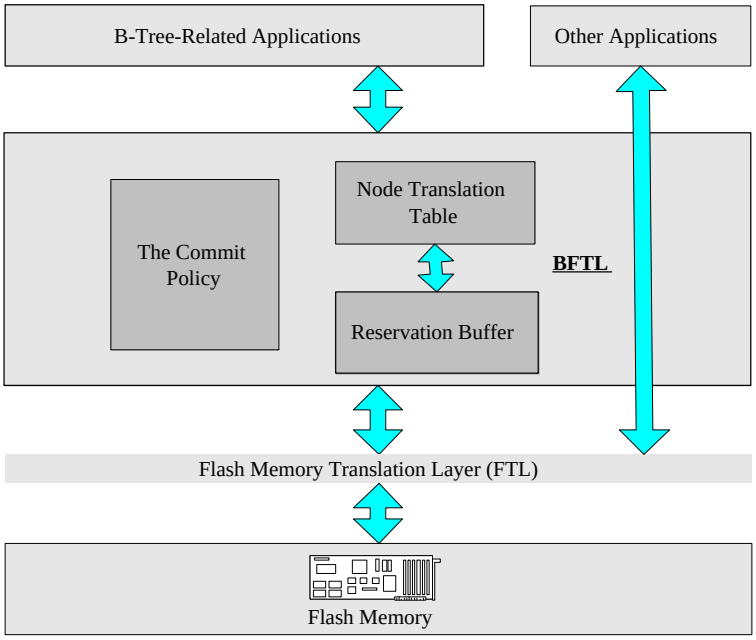


Figure 5: Architecture of a System Which Adopts BFTL.

size of a page), the storing of the index units is handled by a commit policy. Many index units could be smartly packed into few sectors to reduce the number of pages physically written. Note that the "sectors" are logical items which are provided by the block-device emulation of FTL. We would try to pack index units belonging to different B-Tree nodes in a small number of sectors. During this packing process, although the number of sectors to be updated is reduced, index units of one B-Tree node could now exist in different sectors. To help BFTL to identify index units of the same B-Tree node, a node translation table is adopted.

In the following sub-sections, we shall present the functionality of index units, the commit policy, and the node translation table. In Section 3.2 we illustrate how a B-Tree node is physically represented by a collection of index units. The commit policy which smartly flushes the dirty records is presented in Section 3.3. The design issues of the node translation table are discussed in Section 3.4.

3.2 The Physical Representation of a B-Tree Node: The Index Units

When applications insert, delete, or modify records, the dirty records could be temporarily held by the reservation buffer of BFTL. BFTL would construct a corresponding "index unit" to reflect the primary-key insertion/deletion to the B-Tree index structure caused by a dirty record. In other words, an index unit could be treated as a modification of the corresponding B-Tree node, and a B-Tree node could be logically constructed by collecting and parsing all relevant index units. Since the size of a index unit is relatively small (compared to the size of a page), the adopting of index units could prevent redundant data from frequently being written to flash memory. To save space needed by the storing of index units, many index units are packed into few sectors even though the packed index units might be belonging to different B-Tree nodes. As a result, the index units of one B-Tree node could exist in different sectors over flash memory, and the physical representation of the B-Tree node would be different from the original one.

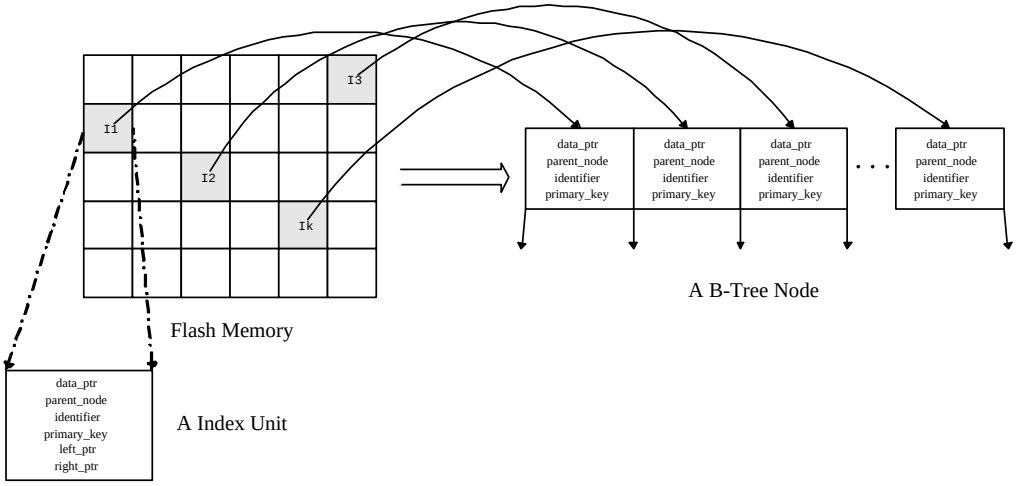


Figure 6: The node consists of index units.

To construct the logical view of a B-Tree node, relevant index units are collected and parsed for the layer above BFTL, i.e., users of BFTL. Figure 6 illustrates how the logical view of a B-Tree node is constructed: Index units (I1, I2, ... , Ik) of a B-Tree node are scattered over flash memory, and we could form the B-Tree node by collecting its relevant index units over flash memory. An index unit is of several components: data_ptr, parent_node, primary_key, left_ptr, right_ptr, an identifier, and an op_flag. Where data_ptr, parent_node, left_ptr, right_ptr, and primary_key are the elements of a original B-Tree node. They represent a reference to the record body, a pointer to the parent B-Tree node, a pointer to the left B-Tree node, a pointer to the right B-Tree node, and and the primary key, respectively. Beside the components originally for a B-Tree node, an identifier is needed: The identifier of an index unit denotes to which B-Tree node the index unit is belonging. The op_flag denotes the operation done by the index unit, and the operations could be an insertion, a deletion, or an update. Additionally, time-stamps are added for each batch flushing of index units to prevent BFTL from using stale index units. Note that BFTL uses FTL to store index units. As shown in Figure 6, index units related to the desired B-Tree node are collected from flash memory. Index units could be scattered over flash memory. The logical view of the B-Tree node is constructed through the help of BFTL. As astute readers might point out, it is very inefficient to scan flash memory to collect the index units of the same B-Tree node. A node translation table is adopted to handle the collection of index units. It will be presented in Section 3.4.

3.3 The Commit Policy

Dirty records are temporarily held by the reservation buffer of BFTL. The buffer should be flushed out in a timely fashion. Index units are generated to reflect modifications to B-Tree index structures, and the index units are packed into few sectors and written to flash memory (by FTL). A technical issue is how to smartly pack index units into few sectors. In this section, we shall provide discussions on commit policies for index units.

The reservation buffer in BFTL is a buffer space for dirty records. The buffering of dirty records could prevent B-Tree index structures over flash memory from being intensively modified. However, the capacity of the reservation buffer is not unlimited. Once the reservation buffer is full, some dirty records in the buffer should be committed (written) to flash memory. We propose to flush out all dirty records in this paper because a better analysis of dirty records is possible to reduce updates of leaf nodes (We will demonstrate the approach later in the

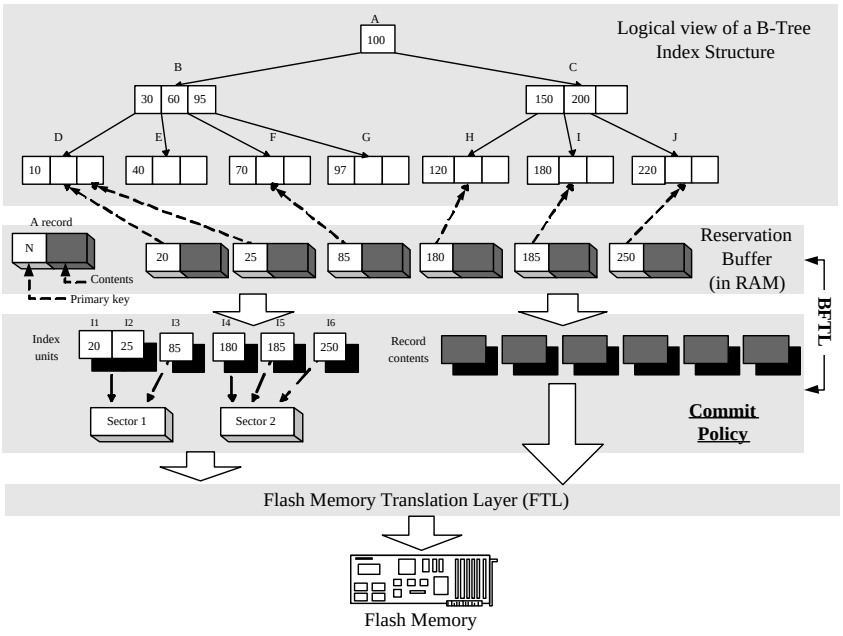


Figure 7: The Commit Policy Packs and Flushes the Index Units.

performance evaluation.) Beside the storing of records, BFTL would construct index units to reflect modifications to the B-Tree index structure. Since the size of an index unit is smaller than the sector size provided by FTL (or the page size of flash memory), many index units should be packed together in order to further reduce the number of sectors needed. On the other hand, we also hope that index units of the same B-Tree node will not be scattered over many sectors so that the collection of the index units could be more efficient. A commit policy is proposed to achieve both of the objectives. We shall illustrate the commit policy by an example:

The handling of a B-Tree index structure in Figure 7 is divided into three parts: the logical view of a B-Tree index structure, BFTL, and FTL. Suppose that the reservation buffer could hold six records whose primary keys are 20, 25, 85, 180, 185, and 250, respectively. When the buffer is full, the records should be written to flash memory. BFTL first generates six index units (I1 to I6) for the six records. Based on the primary keys of the records and the value ranges of the leaf nodes (D, E, F, G, H, I, and J in the figure), the index units could be partitioned into five disjoint sets: $\{\text{I1, I2}\} \in D$, $\{\text{I3}\} \in F$, $\{\text{I4}\} \in H$, $\{\text{I5}\} \in I$, $\{\text{I6}\} \in J$. The partitioning prevents index units of the same B-Tree node from being fragmented. Suppose that a sector provided by FTL could store three index units. Therefore, {I1, I2} and {I3} would be put in the first sector. {I4}, {I5}, and {I6} would be put in the second sector since the first sector is full. Finally, two sectors are written to commit the index units. If the reservation buffer and the commit policy are not adopted, up to six sector writes might be needed to handle the modifications of the index structure.

As astute reader may notice, the packing problem of index units into sectors is inherently intractable. A problem instance is as follows: Given disjoint sets of index units, how to minimize the number of sectors in packing the sets into sectors?

Theorem 1 *The packing problem of index units into sectors is NP-Hard.*

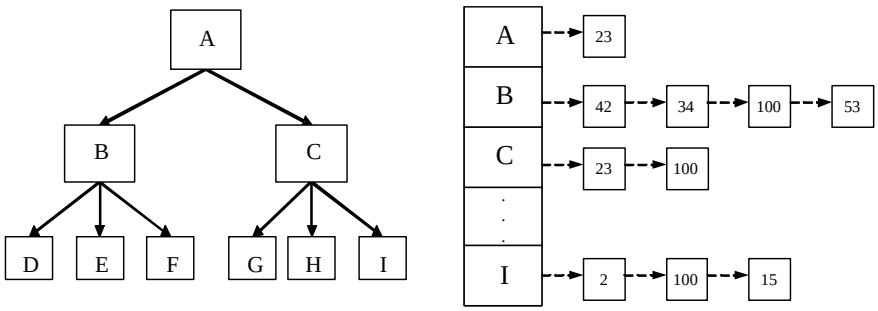
Proof. The intractability of the problem could be shown by a reduction from the Bin-Packing [21] problem:

Let an instance of the Bin-Packing problem be defined as follows: Suppose B and K denote the capacity of a bin and the number of items, where each item has a size. The problem is to put items into bins such that the number of bins is minimized.

The reduction can be done as follows: Let the capacity of a sector be the capacity of a bin B, and each item a disjoint set of index units. The number of disjoint sets is as the same as the number of items, i.e., K. The size of a disjoint set is the size of the corresponding item. (Note that although the sector size is determined by systems, the sector size could be normalized to B. The sizes of disjoint sets could be done in the same ratio accordingly.) If there exists a solution for the packing problem of index units, then the solution is also one for the Bin-Packing problem. □

Note that there exist many excellent approximation algorithms for bin-packing. For example, the well-known FIRST-FIT approximation algorithm [25] could have an approximation bound no more than twice of the optimal solution.

3.4 The Node Translation Table



(a) The logical view of a B-tree (b) The node translation table

Figure 8: The Node Translation Table.

Since the index units of a B-Tree node might be scattered over flash memory due to the commit policy, a node translation table is adopted to maintain a collection of the index units of a B-Tree node so that the collecting of index units could be efficient. This section presents the design and related implementation issues of the node translation table.

Since the construction of the logical view of a B-Tree node requires all index units of the B-Tree node, it must be efficient to collect the needed index units when a B-Tree node is accessed. A node translation table is introduced as an auxiliary data structure to make the collecting of the index units efficient. A node translation table is very similar to the logical address translation table mentioned in Section 2.1, which maps an LBA (the address of a sector) to a physical page number. However, different from the logical address translation table, the node translation table maps a B-Tree node to a collection of LBA's where the related index units reside. In other words, all LBA's of the index units of a B-Tree node are chained after the corresponding entry of the node translation table. In order to form a correct logical view of a B-Tree node, BFTL would visit (read) all sectors where the related index units reside and then construct an up-to-date logical view of the B-Tree node for users of BFTL. The node translation table could be re-built by scanning the flash memory when system is powered-up.

Figure 8.(a) shows a B-Tree with nine nodes. Figure 8.(b) is a possible configuration of the node translation

table. Figure 8.(b) shows that each B-Tree node consists of several index units which could come from different sectors. The LBA's of the sectors are chained as a list after the corresponding entry of the table. When a B-Tree node is visited, we collect all the index units belonging to the visited node by scanning the sectors whose LBA's are stored in the list. For example, to construct a logical view of B-Tree node C in Figure 8.(a), LBA 23 and LBA 100 are read by BFTL (through FTL) to collect the needed index units. Conversely, an LBA could have index units which are belonging to different B-Tree nodes. Figure 8.(b) shows that LBA 100 contains index units of B-Tree nodes B, C, and I. Therefore, when a sector is written, the LBA of the written sector might be appended to some entries of the node translation table accordingly .

The following example which illustrates how BFTL locates a record, as shown in Figure 9:

Step 1: An application issues a read command for accessing a record.

Step 2: If the record could be found in the reservation buffer, then return the record.

Step 3: Otherwise; traverse the whole B-Tree form the root node by the node translation table to search for the record.

Step 4: If the record is found, then return the record.

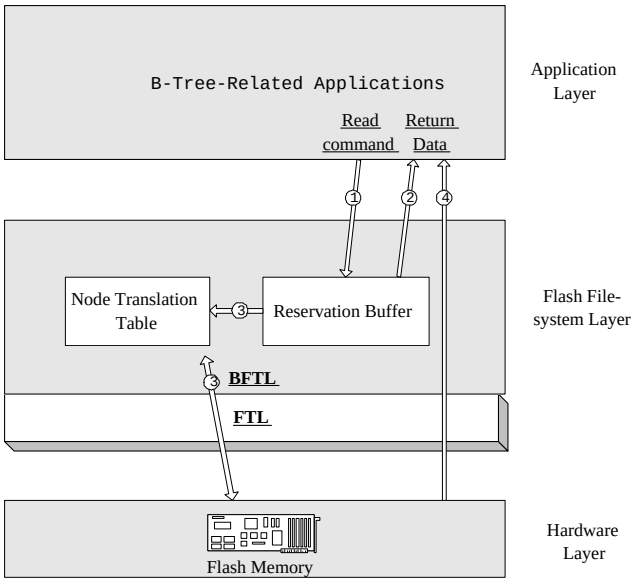


Figure 9: The Procedures to Handle Searching in BFTL.

As astute readers may point out, the lists in the node translation table could grow unexpectedly. For example, if a list after a entry of the node translation table have 100 slots, the visiting of the corresponding B-Tree node might have to read 100 sectors. On the other hand, 100 slots are needed in the node translation table to store the LBA's. If the node translation table is handled in an uncontrolled manner, it will not only deteriorate the performance severely but also consume a lot of resources (such as RAM). To overcome the problem, we propose to compact the node translation table when necessary. A system parameter C is used to control the maximum length of the lists of the node translation table. When the length of a list grows beyond C, the list will be compacted. To compact a list, all related index units are collected into RAM and then written back to

flash memory with a smallest number of sectors. As a result, the size of the table could be bounded by $O(N * C)$, where N denotes the number of B-Tree nodes. On the other hand, the number of sector reads needed to visit a B-Tree node can be bounded by C. Obviously, there is a trade-off between the overheads of compaction and the performance. The experimental results presented in Section 5 could provide more insights for system parameter configuring.

4 System Analysis

This section intends to provide the analysis of the behaviors of BFTL and FTL. We derived the numbers of sectors read and written by FTL and BFTL to handle the insertion of n records.

Suppose that we already have a B-Tree index structure resides on flash memory. Without losing the generality, let a B-Tree node fit in a sector (provided by FTL). Suppose that n records are to be inserted. That is, n primary keys will be inserted into the B-Tree index structure. Assume that the values of the primary keys are all distinct.

First, we shall investigate the behaviors of FTL. A B-Tree node under FTL is stored in exactly one sector. One sector write is needed for each primary key insertion when no node overflow (node splitting) occurs. If a node is overflowed, one primary key in the node will be promoted to its parent node, and the node is then split into two new nodes. The splitting could be handled by three sector writes under FTL. Let H denote the current height of the B-Tree, and N_{split} denote the number of nodes which are split during the handling of the insertions. The numbers of sectors read and written by FTL to handle the insertions could be represented as follows:

$$\begin{cases} R_{FTL} = O(n * H) \\ W_{FTL} = O(n + 2 * N_{split}) \end{cases} \tag{1}$$

Suppose that the sector size remains the same under BFTL (note that BFTL is above FTL), and the hight of the B-Tree is H. Let us consider the numbers of sectors read and written over flash memory when n records are inserted: Because BFTL adopts the node translation table to collect index units of a B-Tree node, the number of sectors that are read to construct a B-Tree node depends on the length of lists of the node translation table. Let the length of the lists be bounded by C (as mentioned in Section 3.4), the number of sectors that are read by BFTL to handle the insertions could be represented as follows: Note that C is a control parameter, as discussed in the previous section.

$$R_{BFTL} = O(n * H * C) \tag{2}$$

Equation 2 shows that the BFTL might read more sectors in handling the insertions. In fact, BFTL trades the number of reads for the number of writes. The number of sectors written by BFTL could be calculated as follows: Because BFTL adopts the reservation buffer to hold records in RAM and flushes them in a batch, modifications to B-Tree nodes (the index units) could be packed in few sectors. Let the capacity of the reservation buffer of a B-Tree be of b records. As a result, the reservation buffer would be flushed by the commit policy at least $\lceil n/b \rceil$ times during the handling of the insertion of n records. Let N^i_{split} denote the number of nodes which are split to handle the i-th flushing of the reservation buffer. Obviously, $\sum_{i=1}^{\lceil n/b \rceil} N^i_{split} = N_{split}$ because the B-Tree index structures under FTL and BFTL are logically identical. For each single step of the reservation buffer flushing, we have $b + N^i_{split} * (fanout - 1)$ dirty index units to commit because the additional $(fanout - 1)$ dirty index units are for the newly created nodes during the splitting, where $fanout$ is the maximum fanout of the B-Tree. Note that N^i_{split} times $(fanout - 1)$ in the formula because each splitting will result in 2 new nodes, and the number

of records in the 2 new nodes is $(fanout - 1)$. Furthermore, the splitting will result in the update of the parent node of the new nodes (that contributes to b in the above formula). Similar to FTL, suppose that a B-Tree node could fit in a sector. That means a sector could hold (fanout-1) index units. Let $\Lambda = (fanout - 1)$. The number of sectors written by the i-th committing of the reservation buffer could be $(\frac{b}{\Lambda} + N^i_{split})$. To completely flush the reservation buffer, we have to write at least $\sum_{i=1}^{\lceil n/b \rceil} (\frac{b}{\Lambda} + N^i_{split}) = (\sum_{i=1}^{\lceil n/b \rceil} \frac{b}{\Lambda}) + N_{split}$ sectors. Since BFTL adopts the FIRST-FIT approximation algorithm (as mentioned in Section 3.3), the number of sectors written by BFTL could be bounded by the following formula:

$$W_{BFTL} = O(2 * (\sum_{i=1}^{\lceil n/b \rceil} \frac{b}{\Lambda}) + N_{split}) = O(\frac{2 * n}{\Lambda} + N_{split}) \tag{3}$$

By putting W_{FTL} with W_{BFTL} together, we have:

$$\begin{cases} W_{BFTL} = O(\frac{2*n}{\Lambda} + N_{split}) \\ W_{FTL} = O(n + 2 * N_{split}) \end{cases} \tag{4}$$

Equation 4 shows that W_{BFTL} is far less than W_{FTL}, since Λ (the number of index units a sector could store) is usually larger than 2. The deriving of equations could provide a low bound for W_{BFTL}. However, we should point out that the compaction of the node translation table (mentioned in Section 3.4) might introduce some run-time overheads. We shall later show that when $\Lambda = 20$, the number of sectors written by BFTL is between 1/3 and 1/13 of the number of sectors written by FTL.

5 Performance Evaluation

The idea of BFTL was implemented and evaluated to verify the effectiveness and to show the benefits of our approach. By eliminating redundant data written to flash memory, we surmise that the performance of B-Tree operations should be significantly improved.

5.1 Experiment Setup and Performance Metrics

A NAND-based system prototype was built to evaluate the performance of BFTL and FTL. The prototype was equipped with a 4MB NAND flash memory, where the performance of the NAND flash memory is included in Table 1. To evaluate the performance of FTL, a B-Tree was directly built over the block-device emulated by FTL. The *greedy* block-recycling policy [1, 15] was adopted in FTL to handle garbage collection.

Because we focused on the behavior of B-Tree index structures in this paper, we did not consider the writing of data records over flash memory. Only the performance of index operations was considered and measured. The fan-out of the B-Tree used in the experiments was 21, and the size of a B-Tree node fits in a sector. To evaluate the performance of BFTL, BFTL was configured as follows: The reservation buffer in the experiments was configured to hold 60 records (unless we explicitly specified the capacity). As suggested by practical experiences in using B-Tree index structure, we assumed that a small amount of B-Tree nodes in the top levels were cached in RAM so that these "hot" nodes could be accessed efficiently. The bound of the lengths of lists in the node translation table was set as 3.

In the experiments, we measured the average response time of record insertions and deletions. A smaller response time denotes a better efficiency in handling requests. The average response time also implicitly reflected

the overheads of garbage collection. If there was a significant number of live page copyings and block erasings, the response time would be increased accordingly. To further investigate the behaviors of BFTL and FTL, we also measured the numbers of pages read, pages written, and blocks erased in the experiments. Note that Sector reads/writes were issued by an original B-Tree index structure or BFTL when BFTL was not adopted or adopted, respectively. FTL translated the sector reads/writes into page reads/writes to physically access the NAND flash memory. Live data copyings and block erases were generated accordingly to recycle free space when needed. Readers could refer to Figure 5 for the system architecture. The energy consumption of BFTL and FTL was measured to evaluate their power-efficiency levels. Different simulation workloads were used to measure the performance of BFTL and FTL. The details will be illustrated in later sections.

5.2 Performance of B-Tree Index Structures Creation

In this part of the experiments, we measured the performance of FTL and BFTL in the creating of B-Tree index structures. B-Tree index structures were created by record insertions. In other words, the workloads consisted of insertions only. For each run of experiments, we inserted 24,000 records. We must point out that although a B-Tree constructed by the 24,000 record insertions under FTL occupied 868KB space on flash memory, however, the amount of total data written by FTL was 14MB. Because a 4MB NAND flash memory was used in the experiments, garbage collection activities would be started to recycle free space. In the experiments, a ratio rs was used to control the value distribution of the inserted keys: When rs equals to zero, that means all of the keys were randomly generated. If rs equals to 1, that means the value of the inserted keys were in an ascending order. Consequently, if the value of rs equals to 0.5, that means the values of one-half of the keys were in an ascending order, while the other keys were randomly generated. In Figure 10.(a) through Figure 10.(e), the X-axes denote the value of rs.

Figure 10.(a) shows the average response time of the insertions. We can see that BFTL greatly outperformed FTL: The response time of BFTL was even one-twentieth of FTL when the values of the keys were completely in an ascending order ($rs = 1$). BFTL still outperformed FTL even if the values of the keys were randomly generated ($rs = 0$). When the keys were sequentially generated ($rs = 1$), the number of sectors written could be decreased because index units of the same B-Tree node would not be scattered over sectors severely. Furthermore, the length of the lists of the node translation table would be relatively short and the compaction of the lists would not introduce significant overheads. As mentioned in the previous sections, writing to flash memory is relative expensive because writes would wear flash, consume more energy, and introduce garbage collection. Figure 10.(b) and Figure 10.(c) show the number of pages written and the number of pages read in the experiments, respectively. The numbers could reflect the usages of flash memory by FTL and BFTL in the experiments. If we further investigate the behaviors of BFTL and FTL by putting Figure 10.(b) with Figure 10.(c) together, we can see that BFTL smartly traded extra reads for the number of writes by the adoption of the commit policy. On the other hand, the extra reads come from the visiting of sectors to construct a logical view of a B-Tree node, as mentioned in Section 3.4.

For the garbage collection issue, in Figure 10.(d) we can see that BFTL certainly suppressed the garbage collection activities when compared with FTL. In some experiments of BFTL, garbage collection even did not start yet. As a result, a longer lifetime of flash memory could be faithfully promised by BFTL. Figure 10.(e) shows the overheads introduced by the compaction of the node translation table. In Figure 10.(e), we can see that the number of executions of compacting was reduced when the values of the inserted keys were in an ascending order. On the other hand, BFTL frequently compacted the node translation table if the values of the inserted keys were randomly generated since the index units of a B-Tree node were also randomly scattered over sectors. Therefore the length of the lists could grow rapidly and the lists would be compacted frequently.

Creation		
	BFTL	FTL
rs=0	11.65	12.94
rs=1	0.931	11.104

Maintainence		
	BFTL	FTL
50/50, rs=0	8.804	8.261
50/50, rs=1	6.136	9.826
10/90, rs=0	10.38	10.99
10/90, rs=1	1.593	9.515

Table 2: Energy Dissipations of BFTL and FTL (*joule*)

5.3 Performance of B-Tree Index Structures Maintenance

In the section, the performance of BFTL and FTL to maintain B-Tree index structures was measured. Under the workloads adopted in this part of experiments, records are inserted, modified, or deleted. To reflect realistic usages of index services, we varied the ratio of the number of deletions to the number of insertions. For example, a 30/70 ratio denotes that the thirty percent of total operations are deletions and the other seventy percent of total operations are insertions. For each run the experiments, 24,000 operations were performed on the B-Tree index structures and the ratio of deletion/insertion was among 50/50, 40/60, 30/70, 20/80, and 10/90. Besides the deletion/insertion ratios, $rs = 1$ and $rs = 0$ (please see Section 5.2 for the definition of rs) were used as two representative experiment settings.

The X-axes of Figure 11.(a) and Figure 11.(b) denote the ratios of deletion/insertion. Figure 11.(a) shows the average response time under different ratios of deletions/insertions. The average response time shows that BFTL outperformed FTL when $rs = 0$ (the keys were randomly generated), and the performance advantage was more significant when $rs = 1$ (the values of the keys were in an ascending order). When $rs = 1$, the performance of BFTL greatly improved when the ratio of deletions/insertions changed from 50/50 to 10/90. For the experiment of BFTL under a 50/50 ratio, because records were frequently inserted and deleted, a lot of index units for insertions and deletions were generated. As a result, BFTL had to visit more sectors to collect the index units of a B-Tree node under a 50/50 ratio than under a 10/90 ratio. Different from those of $rs = 1$, the performance gradually degraded when the ratio changed from 50/50 to 10/90 when $rs = 0$ (random). Since the inserted keys were already randomly generated, a 10/90 ratio denoted more keys were generated and inserted than a 50/50 ratio. As a result, more index units could be chained in the node translation table so that the visiting of a B-Tree node was not very efficient. Figure 11.(b) shows the number of block erased in the experiments. The garbage collection activities were substantially reduced by BFTL, and they had even not started yet in the experiments of $rs = 1$ of BFTL.

5.4 The Size of the Reservation Buffer and the Energy Consumption Issues

In this part of experiments, we evaluated the performance of BFTL under different sizes of the reservation buffer so that we could have more insights in the configuring of the reservation buffer. We also evaluated the energy consumptions under BFTL and FTL. Because BFTL could have a reduced number of writes, energy dissipations under BFTL is surmised to be lower than under FTL.

There is a trade-off to configure the size of the reservation buffer: A large reservation buffer could have benefits from buffering/caching records, however, it could damage the reliability of BFTL due to power-failures.

Reservation buffers with different size were evaluated to find a reasonably good setting. The experiment setups in Section 5.2 were used in this part of experiments, but the value of rs was fixed at 0.5. The size of the reservation buffer was set between 10 records and 120 records, and the size was incremented by 10 records. Figure 12 shows the average response time of the insertions: The average response time was significantly reduced when the size of the reservation buffer was increased from 10 records to 60 records. After that, the average response time was linearly reduced and no significant improvement could be observed. Since further increasing the size of the reservation buffer could damage the reliability of BFTL, the recommended size of the reservation buffer for the experiments was 60 records.

Energy consumption is also a critical issue for portable devices. According to the numbers of reads/ writes/ erases generated in the experiments, we calculated the energy consumption contributed by BFTL and FTL. The energy consumptions of reads/ writes/ eraess are included in Table 1. The calculated energy consumption of the experiments are listed in Table 2: The energy consumed by BFTL was clearly less than FTL. Since page writes and block erases consume relatively more energy than page reads, the energy consumption was reduced when BFTL smartly traded extra reads for the number of writes. Furthermore, energy consumption contributed by garbage collection was also reduced by BFTL since BFTL consumed free space slower than FTL.

6 Conclusion

Flash-memory storage systems are very suitable for embedded systems such as portable devices and consumer electronics. Due to hardware restrictions, the performance of NAND flash memory could deteriorate significantly when files with index structures, such as B-Tree, are stored. In this paper, we propose a methodology and a layer design to support B-Tree index structures over flash memory. The objective is not only to improve the performance of flash-memory storage systems but also to reduce the energy consumption of the systems, where energy consumption is an important issue for the design of portable devices. BFTL is introduced as a layer over FTL to achieve the objectives. BFTL reduces the number of redundant data written to flash memory. We conducted a series of experiments over a system prototype, for which we have very encouraging results.

There are many promising research directions for the future work. With the advance of flash-memory technology, large-scaled flash-memory storage systems could become very much affordable in the near future. How to manage data records and their index structures, or even simply storage space, over huge flash memory might not have a simple solution. The overheads in flash-memory management could introduce a serious performance in system designs.

References

[1] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," USENIX Technical Conference on Unix and Advanced Computing Systems, 1995 .

[2] Aleph One Company, "Yet Another Flash Filing System".

[3] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer: "An Asymptotically Optimal Multiversion B-Tree," VLDB Journal 5(4): 264-275(1996)

[4] Compact Flash Association, "$CompactFlash^{TM}$ 1.4 Specification," 1998.

[5] D. Woodhouse, Red Hat, Inc. "JFFS: The Journalling Flash File System".

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

[7] H. Yokota, Y. Kanemasa, and J. Miyazaki, "Fat-Btree: An Update-Conscious Parallel Directory Structure".

[8] H. Yokota, Y. Kanemasa, and J. Miyazaki: "Fat-Btree: An Update-Conscious Parallel Directory Structure," ICDE 1999: 448-457

[9] Intel Corporation, "Flash File System Selection Guide".

[10] Intel Corporation, "Understanding the Flash Translation Layer(FTL) Specification".

[11] Intel Corporation, "Software Concerns of Implementing a Resident Flash Disk".

[12] Intel Corporation, "FTL Logger Exchanging Data with FTL Systems".

[13] Intel Corporation, "LFS File Manager Software: LFM".

[14] K. Han-Joon, and L. Sang-goo, A New Flash Memory Management for Flash Storage System, Proceedings of the Computer Software and Applications Conference, 1999.

[15] L. P. Chang, T. W. Kuo,"A Real-time Garbage Collection Mechanism for Flash Memory Storage System in Embedded Systems," The 8th International Conference on Real-Time Computing Systems and Applications (RTCSA 2002), 2002.

[16] L. P. Chang, and T. W. Kuo, "A Dynamic-Voltage-Adjustment Mechanism in Reducing the Power Consumption of Flash Memory for Portable Devices," IEEE Conference on Consumer Electronic (ICCE 2001), LA. USA, June 2001.

[17] L. P. Chang, and T. W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," The 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002) September 24 V 27, 2002. San Jose, California.

[18] M. Rosenblum, and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems 10(1) (1992) pp.26-52.

[19] M. Wu, and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1994), 1994.

[20] M. Freeston, "A General Solution of the n-dimensional B-Tree Problem," SIGMOD Conference, San Jose, May 1995.

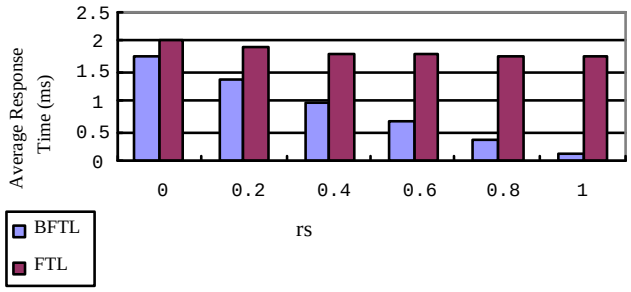
[21] M. R. Garey, and D. S. Johnson, "Computers and intractability", 1979.

[22] R. Bayer, and E. M. McCreight: "Organization and Maintenance of Large Ordered Indices," Acta Informatica 1: 173-189(1972).

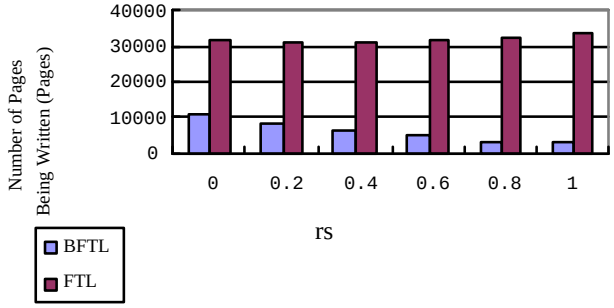
[23] SSFDC Forum, "$SmartMedia^{TM}$ Specification", 1999.

[24] T. W. Kuo, J. H. Wey, and K. Y. Lam, "Real-Time Data Access Control on B-Tree Index Structures," the IEEE 15th International Conference on Data Engineering (ICDE 1999), Sydney, Australia, March 1999.

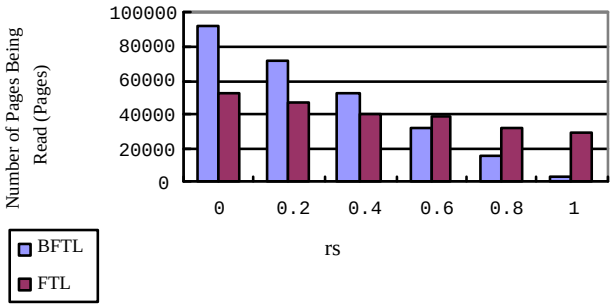
[25] Vijay V. Vazirani, "Approximation Algorithm," Springer publisher, 2001.



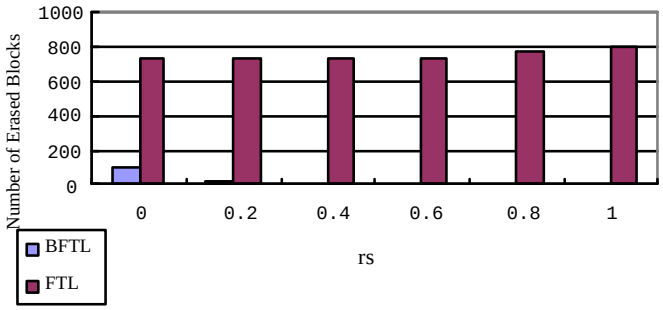
(a) Average Response Time of Insertion after Inserting 24,000 Records



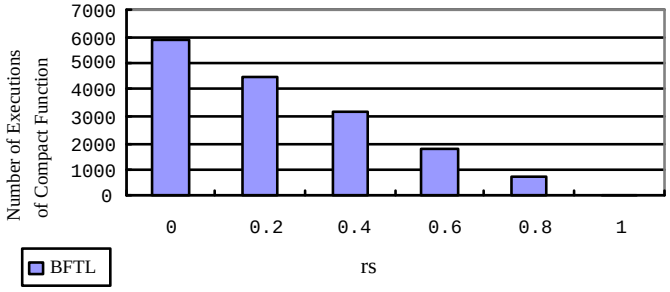
(b) Number of Pages Being Written after Inserting 24,000 Records



(c) Number of Pages Being Read after Inserting 24,000 Records

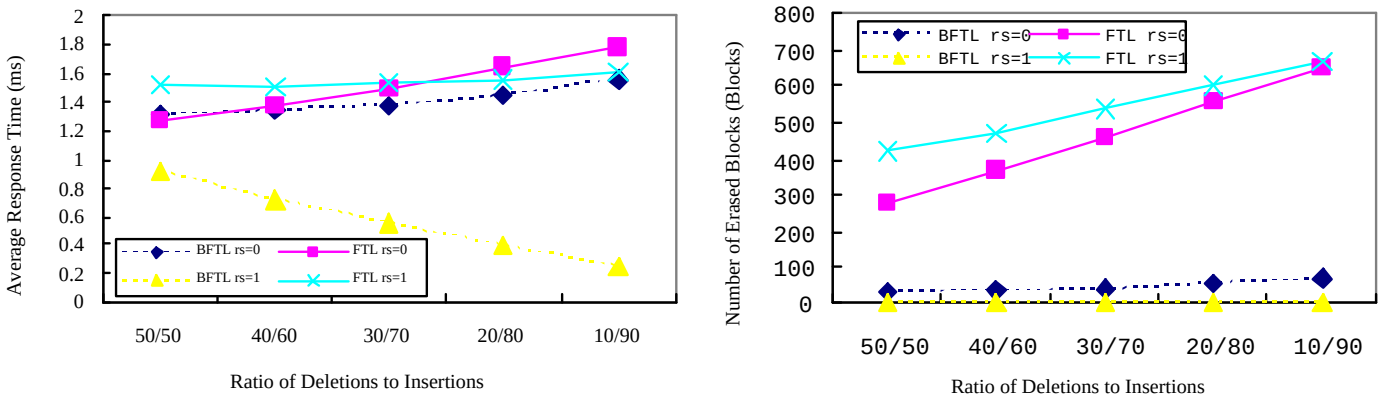


(d) Number of Erased Blocks after Inserting 24,000 Records



(e) Number of Executions of Compact Function after Inserting 24,000 Records

Figure 10: Experimental Results of B-Tree Index Structures Creation.



(a) Average Response Time under Different Ratios of Deletions/Insertions

(b) Number of Block Erased under Different Ratios of Deletions and Insertions

Figure 11: Experimental Results of B-Tree Index Structures Maintenance.

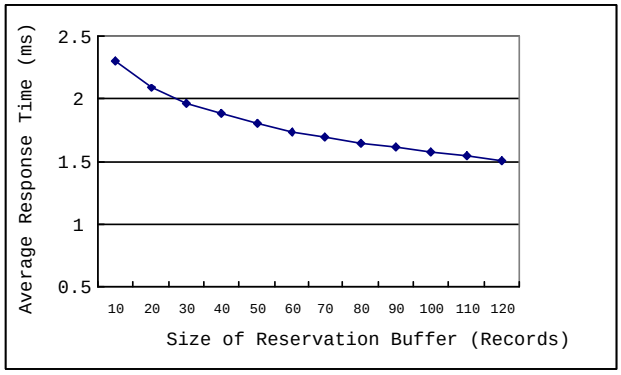


Figure 12: Experimental Results of BFTL under Different Sizes of the Reservation Buffer