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

CHIN-HSIEN WU and TEI-WEI KUO
National Taiwan University
and
LI PING CHANG
National Chiao-Tung University

With the significant growth of the markets for consumer electronics and various embedded systems, flash memory is now an economic solution for storage systems design. Because index structures require intensively fine-grained updates/modifications, block-oriented access over flash memory could introduce a significant number of redundant writes. This 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 can 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); hence, no modifications to existing application systems are needed. We demonstrate that when index structures are adopted over flash memory, the proposed methodology can significantly improve the system performance and, at the same time, reduce both the overhead of flash-memory management and the energy dissipation. The average response time of record insertions and deletions was also significantly reduced.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; H.3.1 [**Content Analysis and Indexing**]: Indexing Methods; H.3.3 [**Information Search and Retrieval**]: Search Process

General Terms: Design, Performance, Algorithm

Additional Key Words and Phrases: Flash memory, B-tree, storage systems, embedded systems, database systems

ACM Reference Format:

Wu, C.-H., Kuo, T.-W., and Chang, L.-P. 2007. An efficient B-tree layer implementation for flash-memory storage systems. ACM Trans. Embedd. Comput. Syst. 6, 3, Article 19 (July 2007), 23 pages. DOI = 10.1145/1275986.1275991 http://doi.acm.org/ 10.1145/1275986.1275991

Authors' addresses: Chin-Hsien Wu and Tei-Wei Kuo, Department of Computer Science and Information Engineering, Graduate Institute of Networking and Multimedia National Taiwan University, Taipei, Taiwan, ROC; email: d90003@csie.ntu.edu.tw; ktw@csie.ntu.edu.tw; Li Ping Chang, Department of Computer and Information Science, National Chiao-Tung University, Hsin Chu, Taiwan, ROC; email: lpchang@cis.nctu.edu.tw.

1. INTRODUCTION

Flash memory is a popular alternative for the design of storage systems because of its shock-resistant, energy-efficient, and nonvolatile nature. In recent years, flash-memory technology has advanced along with the wave of consumer electronics and embedded systems. There are significant technology breakthroughs in both capacity and reliability. The ratio of cost to capacity has being increasing dramatically. Now, a 16-GB NAND flash-memory chip (SAMSUNG K9WAG08U1M flash memory chips) is currently on the market. Flash memory could be considered as an alternative to hard disks in many applications. Now, the implementation of index structures, which are very popular in the organization of data on disks, must be considered regarding flash memory. However, with the very distinctive characteristics of flash memory, traditional designs of index structures result in the severe performance degradation of a flash-memory storage system, significantly reducing 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[Woodhouse], LFM[b:L], and YAFFS [b:Y] were proposed to directly manage raw flash memory. The file systems under this approach are very similar to the log-structured file systems (LFS) [Rosenblum and Ousterhout 1992]. 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 easily built over a emulated block device. For example, FTL/FTL-Lite [b:F b], [b:R], [b:F a], CompactFlash [b:C 1998], and SmartMedia [b:S 1999] are popular and transparent block-device emulations for flash memory. Regardless of which approach is adopted, they share 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 growth of its capacity, the implementations of index structures could become a bottleneck in the performance of flash-memory storage systems. In particular, B-tree is one of the most popular index structures for data accessing because of its scalability and efficiency, where a binary tree is a reduced version of B-tree with a smaller set of search keys and child pointers. We know B-tree is not only used in many database systems [Ramakrishnan and Gehrke 2003] (e.g., IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE), but also some journal file systems (e.g., XFS, JFS, and ReiserFS *)[b:b]. B-tree indices were first introduced by Bayer and McCreight [1972]. Comer [1979] later proposed a variation called B+−tree indices. B-tree index structures are extended to many application domains. Kuo et al. [1999] demonstrated how to provide a predictable performance with B-tree. Freeston [1995] showed multidimensional B-trees, which have good predictable and controllable worst-case characteristics. For the parallel environment, Yokota et al. [1999] proposed Fat-B-trees to improve high-speed access for parallel database systems. Becker et al. [1996] improved the availability of data by a multiversion index structure that supports insertions, deletions, range queries, and exact match queries for the current or some past versions.

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

In this paper, we focus on an efficient integration of B-tree index structures (because of their popularity and practicability) and the block-device emulation mechanism provided by FTL (flash translation layer). We propose a layer which provides a B-tree index management over flash-memory storage systems to handle intensive byte-wise operations because of B-tree access. The layer, referred to as BFTL, could be adopted or removed as needed. In the baseline solution, BFTL is introduced as a layer between file systems and FTL. BFTL could be adopted or removed as needed, where BFTL provides extra considerations for B-tree implementations in eliminating overhead as a result of B-tree index manipulations. The implementation is done transparently over FTL so that no modifications to the existing B-tree-related applications are needed. The overhead of intensive byte-wise operations are caused by record inserting, record deleting, and B-tree reorganizing. For example, the insertion of a record results in the insertion of a data pointer at a leaf node and, possibly, the insertion of tree pointers in the B-tree. Such actions 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 significantly improves the system performance and, at the same time, reduces the overhead of flash-memory management and the energy dissipation, when index structures are adopted for flash memory. We must point out that log-structured file systems over flash memory (i.e., YAFFS/JFFS) would result in similar overhead and BFTL could be easily extended and adopted in log-structured file systems.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 provides an overview of flash memory and discussions of the implementation problems of B-tree over flash memory. Section 4 introduces our approach and its implementation. Section 5 provides performance analysis of the approach. Section 6 shows experimental results. Section 7 is the conclusion and future work.

2. RELATED WORK

In recent years, issues of flash memory management have drawn a lot of attention. Excellent research results and implementations were reported on performance enhancement, especially on garbage collection and system architecture designs [Kawaguchi et al. 1995; Kim and Lee 1999; Chang and Kuo 2002; Wu and Zwaenepoel 1994; Chang et al. 2004; Wu et al. 2004, 2006a, 2006b; Kim et al. 2002; Douglis et al. 1994; Han-Joon and Sang-goo 1999; Chang and Kuo 2001; Park et al. 2004, 2003]. In particular, Kawaguchi, et al. proposed a *cost–benefit* policy [Kawaguchi et al. 1995] with a value-driven heuristic function for block recycling. Kim and Lee [1999] proposed to periodically move live data among blocks so that blocks have more even erase counts. Chang and Kuo [2002] considered an adaptive striping architecture for multiple banks for performance enhancement. Wu and Zwaenepoel [1994] proposed to adopt SRAM as write buffers and presented several cleaning policies for garbage collection. Chang and Kuo [Chang et al. 2004] introduced a real-time garbage collection mechanism to provide QoS guarantees for performance-sensitive applications. Wu et al. [2004] proposed an interrupt-emulation mechanism to reduce the interference of I/O activities on the executions of user tasks such that the entire system performance is improved. Wu et al. [2006a] proposed a method for efficient initialization and crash recovery over flash-memory file systems. Wu, Kuo, and Yang [2006b] proposed a space-efficient search-tree like data structure to accelerate the matching of a given logical address and its corresponding physical address on flash memory. Kim et al. [2002] proposed a space-efficient translation layer for compact-flash systems with reasonable memory usage. Douglis et al. [1994] investigated the performance of some alternative storage devices (i.e., hard disks and flash memory) for mobile computers. Kwoun et al. [Han-Joon and Sang-goo 1999] proposed to periodically move live data among blocks so that blocks have more and even lifetime. Because flash memory also contributes a significant portion of energy consumption, Chang and Kuo [2001] introduced an energy-efficient request-scheduling algorithm for flash-memory storage system to lengthen the operating time of battery-powered portable devices. Park et al. [2004] and Park et al. [2003] presented an energy-aware demand paging technique to lower the energy consumption of embedded systems and also proposed a low-cost memory architecture, which incorporates NAND flash memory into an existing memory hierarchy, for code execution. However, when database or information-processing applications for flash-memory storage systems are considered, the index processing to resolve performance problems caused by intensive byte-wise updates becomes important. As far as we know, no previous work has been done in resolving B-tree implementation problems over flash-memory storage systems.

3. MOTIVATION

In this section, we shall briefly introduce the characteristics of flash memory. The motivation of this work is to show the very distinctive properties of flash memory and to address the potential issues of building a B-trees index structure over NAND flash memory.

3.1 Flash Memory Characteristics

NAND[1] flash memory is organized in 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 NAND flash memory is 16 KB and 512 bytes, respectively. Because flash memory is write-once, we do not overwrite data on each 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 "outplace update." In other words, any existing data on flash memory could not be overwritten (updated) unless it is erased. The pages storing live data and dead data are called "live pages" and "dead pages," respectively. Because outplace 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 LBA usually denotes the logical address of a read/write unit in the flash memory. 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 is rebuilt by scanning the flash memory.

After a certain number of page writes, free space on flash memory would be low. Activities then start, which consists of a series of read/write/erase with the intention to reclaim free spaces. 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 become free pages after erasings. How to efficiently choose which blocks should be erased is the responsibility of a *block-recycling policy*. The block-recycling policy tries to reduce the overhead of garbage collection (caused by live data copyings). Under current technology, a flash-memory block has a limitation on the erase cycle count. For example, a block of typical NAND flash memory can be erased 1 million (10^6) times. After that, a worn-out block can suffer from frequent write errors. A "wear-leveling" policy intends to erase all blocks on flash memory evenly, so that a longer overall lifetime is achieved. Obviously, wear-leveling activities impose significant overhead to the flash-memory storage system if the access patterns try to frequently update some specific data.

3.2 Problem Definition

While the storage capacity of flash memory keeps increasing, many systems, including mobile devices (e.g., PDA's, smart phones, Sony VGN-FS500P12 laptops, and IBM X40 laptops), now have flash memory for (additional) secondary storage devices. Because B-tree is one of the most popular data structures adopted by database applications or databases systems (such as IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE), applications with file-processing needs (such as those with index sequential files), or even many

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

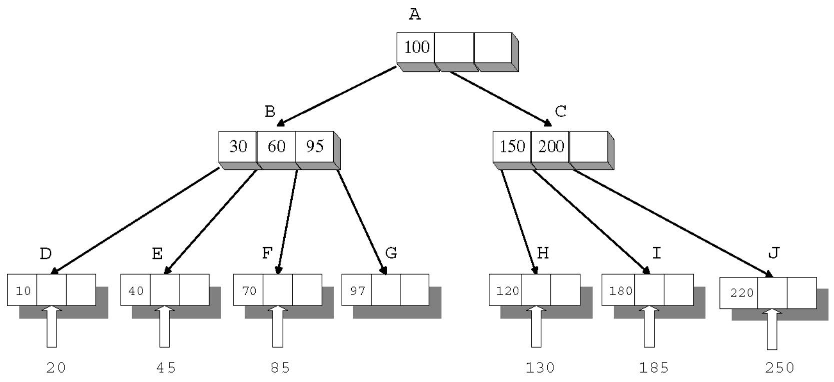


Fig. 1. A B-tree (fanout is four).

well-known journaling file systems (such as XFS, JFS, and ReiserFS *), many applications that run over disks in the past might now access data stored on flash memory. Because of the significant overhead in the manipulations of B-trees ver flash memory, many applications that adopt B-trees will have serious performance problems in accessing data on flash memory. Such an observation motivates this research.

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, B-tree is implemented in disk-storage systems for reducing the I/O fetchings due to B-tree's bigger fanout. As the size of flash memory rapidly grows, the flash memory could replace the disk-storage systems. However, a direct adoption of B-tree index structures over flash-memory storage systems could exaggerate the overhead of flash-memory management. Let us first consider usual operations done over B-tree index structures: Figure 1 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 1, the 1st, 2nd, 3rd, 4th, 5th, and 6th records are 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, because 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 are needed.

Compared with operations on hard disks, updating (or writing) data over flash memory is a very complicated and expensive operation. Since outplace update is adopted, a whole page (512 bytes), which contains the new version of data is written to flash memory and previous data must be invalidated. The page-based write operations introduce a sequence of negative effects. Free

Table I. Performance of a Typical NAND Flash Memory [b:s][a]

	Page Read 512 B	Page Write 512 B	Block Erase 16 KB
Performance(μs)	348	909	1,881
Energy Consumption(μ*joule*)	99	237.6	422.4

[a]Samsung K9F6408U0A 8MB NAND flash memory.

space on flash memory is consumed very quickly. As a result, garbage collection happens frequently to reclaim free space. Furthermore, because flash memory is frequently erased, the lifetime of the flash memory is reduced. Another problem is energy consumption. Outplace updates 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 I, outplace updates eventually consume more energy. For portable devices, because the amount of energy provided by batteries is limited, energy-saving is a major concern. In this paper, we will propose a layer implementation over existing flash memory implementations, i.e., FTL, to not only provide a compatible solution to existing systems, but also present an implementation design to significantly reduce overhead because of B-tree manipulations.

4. 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 of reducing the redundant data written because of the hardware restriction of NAND flash memory. We illustrate the architecture of a system, which adopts BFTL, and present the functionalities of the components inside BFTL in the following subsections.

4.1 Overview

In our approach, we propose a layer we have called BFTL (an efficient B-tree layer for flash-memory storage systems) over the original flash translation layer (FTL). BFTL is devoted to the efficient implementation of B-tree index structures over FTL and provides file systems functions to create and maintain B-tree index structures. In this paper, BFTL is considered as a part of the operating system. Figure 2 illustrates the architecture of a system that 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 from file systems to 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 a corresponding "index unit" for each dirty record. Index units are used to reflect primary-key insertions and deletions to the B-tree index structure caused by the

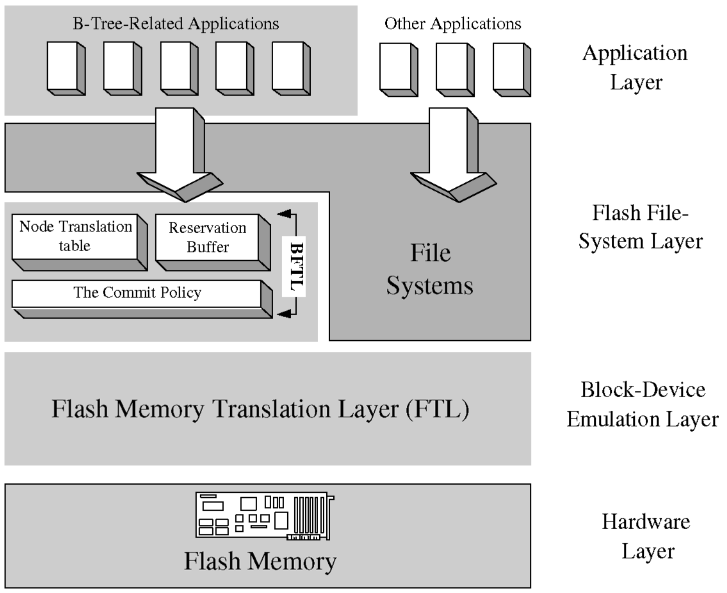


Fig. 2. Architecture of a system which adopts BFTL.

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 allocated (or the original) locations. On the other hand, because an index unit is very small (compared with the size of a page), the storing of the index units is handled by a commit policy. Many index units could be efficiently packed into a 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 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 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 subsections, we present the functionality of index units, the commit policy, and the node translation table. In Section 4.2, we illustrate how a B-tree node is physically represented by a collection of index units. The commit policy, which efficiently flushes the dirty records, is presented in Section 4.3. The design issues of the node translation table are discussed in Section 4.4.

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

BFTL constructs 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 is treated as a modification of the corresponding B-tree node, and a B-tree node is 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 prevents 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 a 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 are different from the original one.

To construct the logical view of a B-tree node, relevant index units are collected and parsed. An index unit has 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 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 belongs. The op_flag denotes the operation done by the index unit. The operation could be an insertion, a deletion, or an update. In addition, 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. Index units could be scattered over flash memory. The logical view of the B-tree node is constructed through the help of BFTL. However, scanning flash memory to collect the index units of the same B-tree node is very inefficient. A node translation table is adopted to handle the collection of index units and is presented in Section 4.4.

4.3 A Commit Policy

A technical issue is how to efficiently pack index units into a few sectors. In this section, we shall provide discussions on a commit policy for index units. First, a reservation buffer for the commit policy is defined as follows: The reservation buffer is a write buffer residing in main memory. When a B-tree node is inserted, deleted, or modified, any newly generated records would, first, be temporarily held by the reservation buffer. Records in the reservation buffer represent operations which have not yet been applied to a B-tree. For each record r in the reservation buffer, there exists a corresponding B-tree node to which r belongs. The relationship of records in the reservation buffer and B-tree nodes is maintained for the commit policy.

The buffering of dirty records prevents B-tree index structures on 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 are committed (written) to flash memory. We propose to flush out all dirty records because a better analysis of dirty records is possible in order to reduce updates of leaf nodes (We will demonstrate the approach later in the performance evaluation.) Beside the storing of records, BFTL constructs index units to reflect modifications to the B-tree index structure. 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 is more efficient. A commit policy is proposed to achieve both of the objectives. The commit policy is illustrated by an example:

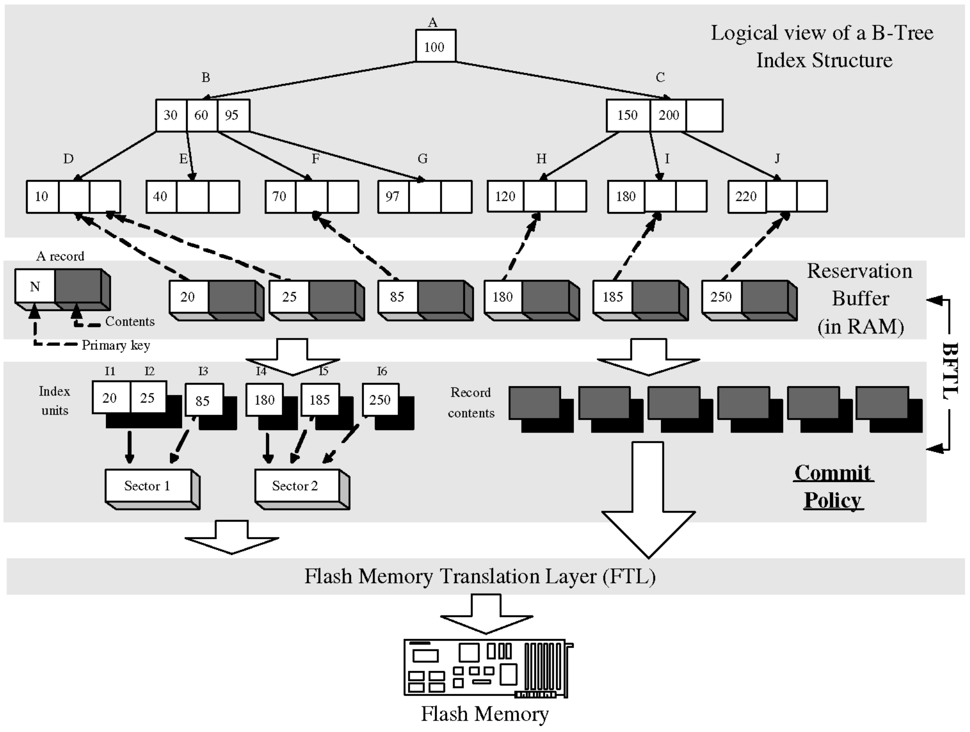


Fig. 3. The commit policy packs and flushes the index units.

The handling of a B-tree index structure in Figure 3 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–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: {I1, I2} $\in D$, {I3} $\in F$, {I4} $\in H$, {I5} $\in I$, {I6} $\in J$. The partitioning prevents index units of the same B-tree node from being fragmented. Suppose that a sector provided by FTL stores three index units. Therefore, {I1, I2} and {I3} are put in the first sector and {I4}, {I5}, and {I6} are 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.

However, the packing problem of index units into sectors is inherently intractable. A problem instance is as follows: given disjoint sets of index units, how are the number of sectors minimized in packing the sets into sectors?

Theorem 4.3.1. *The packing problem of index units into sectors is NP-Hard.*

Proof. The intractability of the problem is shown by a reduction from the bin packing [Garey and Johnson 1979] 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 is 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 is normalized to B. The sizes of disjoint sets are, accordingly, in the same ratio.) If there exists a solution for the packing problem of index units, then the solution is also one for the bin-packing problem. □

Algorithm 1. The FIRST-FIT-based Commit Policy

```
1: Let $\Phi$ denote the set of the disjoint sets of index units
2: Let $\Theta$ denote the set of the sectors
3: while $\Phi$ is not empty do
4:     Let ds be a disjoint set in $\Phi$
5:     if there exists a used sector sec in $\Theta$ that has available free space for ds then
6:         ds is stored in sector sec
7:     else
8:         create a new sector nsec to store ds
9:         $\Theta \leftarrow \Theta + nsec$
10:    end if
11:    $\Phi \leftarrow \Phi - ds$
12: end while
13: flush out $\Theta$ to flash memory
```

There exists many excellent approximation algorithms for bin packing. We propose to adopt the well-known FIRST-FIT approximation algorithm [Vazirani 2001] so as to bound the number of pages written, because of the policy. The pseudocode of the FIRST-FIT-based commit policy is shown in Algorithm 1. Here, let a B-tree node fit in a sector so that the size of a disjoint set is not beyond that of a sector.

4.4 The Node-Translation Table

Since the index units of a B-tree node are scattered over flash memory because of 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 is 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, collecting the needed index units must be efficient 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 3.1, which maps a 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

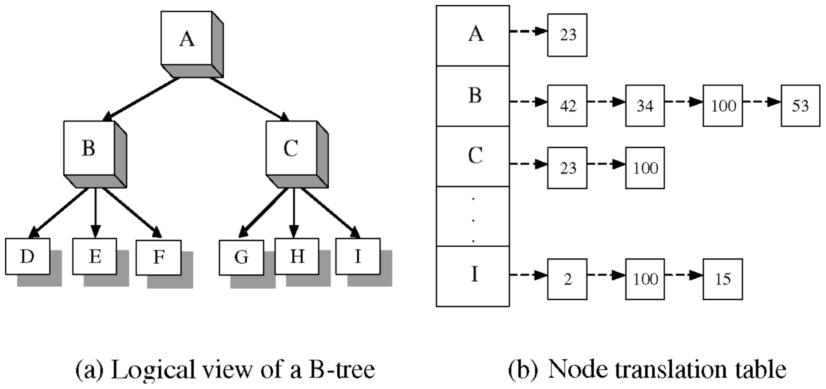


Fig. 4. The node translation table.

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 visits (reads) all sectors where the related index units reside and then construct an up-to-date logical view of the B-tree node. The node-translation table is rebuilt by scanning the flash memory when the system is powered-up.

Figure 4a shows a B-tree with nine nodes. Figure 4b is a possible configuration of the node-translation table. Figure 4b shows that each B-tree node consists of several index units, which 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, all the index units belonging to the visited node are collected 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 4a, LBA 23 and LBA 100 are read by BFTL (through FTL) to collect the needed index units. Conversely, a LBA has index units, which belong to different B-tree nodes. Figure 4b 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 is appended accordingly to some entries of the node-translation table.

However, the lists in the node-translation table can grow unexpectedly. For example, if a list after a entry of the node-translation table has 100 slots, the visiting of the corresponding B-tree node might read 100 sectors. On the other hand, the node-translation table stores 100 LBA's in the list. If the node-translation table is handled in an uncontrolled manner, it will not only severely deteriorate the performance, but also consume many 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 is compacted. To compact a list, all related index units are collected into RAM and then written back to flash memory with the smallest number of sectors. As a result, the size of the table is bounded by $O(N * C)$, where N denotes the number of B-tree nodes. On the other hand, the number of

Algorithm 2. A Revised Commit Policy with the Considerations of the Node-Translation Table

```
 1: Let $\Phi$ denote the set of the disjoint sets of index units
 2: Let $\Theta$ denote the set of the sectors
 3: Let $ntt$ denote a node-translation table
 4: while $\Phi$ is not empty do
 5:   Let $ds$ be a disjoint set in $\Phi$
 6:   Let $en$ be the corresponding entry of $ntt$ of a B-tree node that $ds$ would update
 7:   if the length of the list of the corresponding entry $en$ is beyond $C$ then
 8:     execute the compaction of the list
 9:   end if
10:   if there exists a used sector $sec$ in $\Theta$ that has available free space for $ds$ then
11:     $ds$ is stored in sector $sec$
12:     record the LBA of $sec$ in the list after the corresponding entry $en$ of $ntt$
13:   else
14:     create a new sector $nsec$ to store $ds$
15:     record the LBA of $nsec$ in the list after the corresponding entry $en$ of $ntt$
16:     $\Theta \leftarrow \Theta + nsec$
17:   end if
18:   $\Phi \leftarrow \Phi - ds$
19: end while
20: flush out $\Theta$ to flash memory
```

sector reads needed to visit a B-tree node can be bounded by C. Obviously, there is a trade-off between the overhead of compaction and of performance. The experimental results presented in Section 6 provide more insights for system parameter configuring. The pseudocode of the commit policy is revised to handle the operation of the node-translation table, as shown in Algorithm 2.

5. SYSTEM ANALYSIS

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

Suppose that we already have a B-tree index structure residing 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 overflows, one primary key in the node is promoted to its parent node, and the node is then split into two new nodes. On the other hand, if a node is not half full, the node could be merged with other half-full sibling nodes or rotated one primary key from the other sibling nodes. The splitting is handled by three sector writes under FTL (Two sector writes are for the two new nodes and one sector write is for the parent node). The merging or rotating is also handled by, at most, three sector writes under FTL. Let H denote the current height of the B-tree, and N_{split} and $N_{merge/rotate}$ denote the number of nodes which are split and merged/rotated during the handling of the insertions, respectively. The numbers of sectors read and written by FTL to handle the insertions could be represented as follows:

$$\begin{cases} R_{FTL} = O(n * H) \\ W_{FTL} = O(n + 3 * N_{split} + 3 * N_{merge/rotate}) \end{cases} \tag{1}$$

Suppose that the sector size remains the same under BFTL (note that BFTL is above FTL), and the height 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 4.4), 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 reads 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 are 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 (i.e., the index units) are packed in a few sectors. Let the capacity of the reservation buffer of BFTL be of b records. As a result, the reservation buffer is 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} and $N^i_{merge/rotate}$ denote the number of nodes, which are split and merged/rotated, to handle the ith flushing of the reservation buffer, respectively. Obviously, $\sum_{i=1}^{\lceil n/b \rceil} N^i_{split} = N_{split}$ and $\sum_{i=1}^{\lceil n/b \rceil} N^i_{merge/rotate} = N_{merge/rotate}$ because the B-tree index structures under FTL and BFTL are logically identical. For each single step of the reservation buffer flushing, we have $b + N^i_{split} * fanout + N^i_{merge/rotate} * fanout$ dirty index units to commit. N^i_{split} times *fanout* in the formula because each splitting results in an updating of the parent node and two new nodes in which the number of index units required is *fanout*. Then, $N^i_{merge/rotate}$ times *fanout* in the formula because each merging/rotating results in *fanout* index units, at most (i.e. each merging results in a full new node and an updating of its parent node in which the number of index units required is *fanout*). Similar to FTL, suppose that a B-tree node fits in a sector. That means a sector holds (fanout-1) index units. Let $\Lambda = (fanout - 1)$. The number of sectors written by the ith committing of the reservation buffer is about $(\frac{b}{\Lambda} + N^i_{split} + N^i_{merge/rotate})$. To completely flush the reservation buffer, we have to write at least $\sum_{i=1}^{\lceil n/b \rceil} (\frac{b}{\Lambda} + N^i_{split} + N^i_{merge/rotate}) = (\sum_{i=1}^{\lceil n/b \rceil} \frac{b}{\Lambda}) + N_{split} + N_{merge/rotate}$ sectors. Since BFTL adopts the FIRST-FIT approximation algorithm (as mentioned in Section 4.3), the number of sectors written by BFTL is bounded by the following formula:

$$\begin{aligned} W_{BFTL} &= O\left(2 * \left(\sum_{i=1}^{\lceil n/b \rceil} \frac{b}{\Lambda} + N_{split} + N_{merge/rotate}\right)\right) \\ &= O\left(\frac{2 * n}{\Lambda} + 2 * N_{split} + 2 * N_{merge/rotate}\right) \end{aligned} \tag{3}$$

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

$$\begin{cases} W_{BFTL} = O\left(\frac{2 * n}{\Lambda} + 2 * N_{split} + 2 * N_{merge/rotate}\right) \\ W_{FTL} = O(n + 3 * N_{split} + 2 * N_{merge}) \end{cases} \tag{4}$$

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 two. The deriving of equations provide a low bound for W_{BFTL}. However, the compaction of the node translation table (mentioned in Section 4.4) introduces some run time overhead. We later show that when $\Lambda = 20$, the number of sectors written by BFTL is between 1/5 and 1/26 of the number of sectors written by FTL.

6. PERFORMANCE EVALUATION

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 is significantly improved.

6.1 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 4 MB NAND flash memory, where the performance of the NAND flash memory is included in Table I. To evaluate the performance of FTL, a B-tree was directly built over the block-device emulated by FTL. The *greedy* block-recycling policy [Kawaguchi et al. 1995; Chang et al. 2004] was adopted in FTL to handle garbage collection.

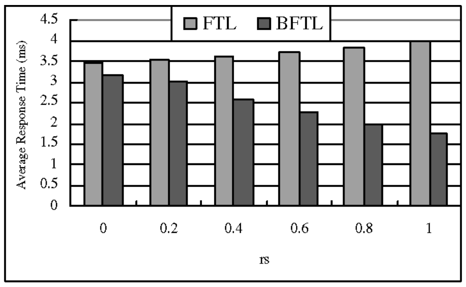
Because we focused on the behavior of B-tree index structures in this paper, we did not consider the writing of data records over flash memory. Only the performance of index operations was considered and measured. The fanout of the B-tree used in the experiments was 21 and the size of a B-tree node fits 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) and the bound of the lengths of lists in the node translation table was set as four (unless we explicitly specified the bound).

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 overhead 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 and BFTL. 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 can refer to Figure 2 for the system architecture.) The energy consumption of BFTL and FTL were measured to evaluate their power-efficiency levels. Different workloads were used to measure the performance of BFTL and FTL. The details are illustrated in later sections.

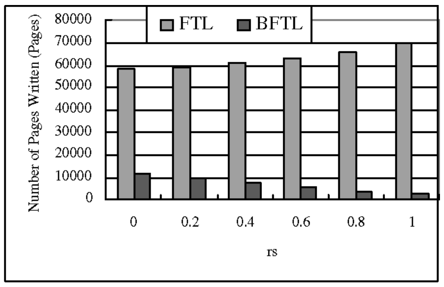
6.2 Performance of B-Tree Index Structures Creation

In this part of 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 30,000 records. Although a B-tree constructed by the 30,000 record insertions under FTL occupied 1197 KB space on flash memory, the total amount of data written by FTL was 14 MB. Because 4 MB NAND flash memory was used in the experiments, the activities of garbage collection were started soon to recycle free space. Note that the rationale behind the experiment setup of 4 MB NAND flash memory was to provide better observations on garbage collection behaviors. In the experiments, a ratio *rs* was used to control the value distribution of the inserted keys: when *rs* equals zero, all of the keys were randomly generated. If *rs* equals 1, the value of the inserted keys were in an ascending order. Consequently, if the value of *rs* equals 0.5, the values of one-half of the keys were in an ascending order, while the other keys were randomly generated. In Figures 5 and 6, the x axes denote the value of *rs*.

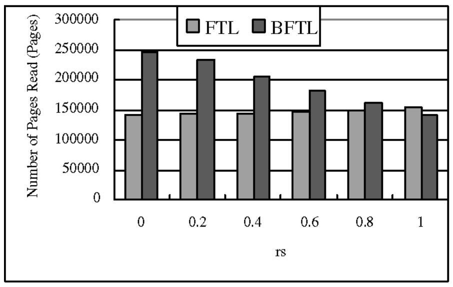
Figure 5a shows the average response time of the insertions. We can see that BFTL greatly outperformed FTL: The response time of BFTL was about one-third of FTL when the values of the keys were completely in an ascending order ($rs = 1$). BFTL still outperformed FTL even if the values of the keys were randomly generated ($rs = 0$). When the keys were sequentially generated ($rs = 1$), the number of sectors written was decreased because index units of the same B-tree node were not severely scattered over sectors. 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. Figures 5b and 5c show the number of pages written and the number of pages read in the experiments, respectively. The numbers 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 5b together with 5c, we can see that BFTL efficiently traded extra reads for the number of writes by adopting 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 4.4.



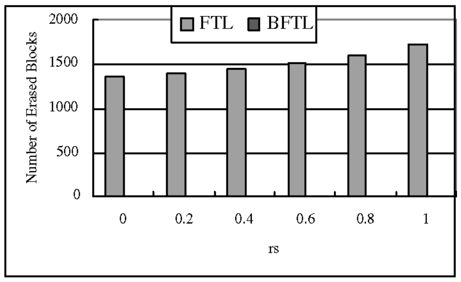
(a) Average Response Time of Insertion after Inserting 30,000 Records



(b) Number of Pages Written after Inserting 30,000 Records



(c) Number of Pages Read after Inserting 30,000 Records



(d) Number of Erased Blocks after Inserting 30,000 Records

Fig. 5. Experimental results of B-tree index structures creation.

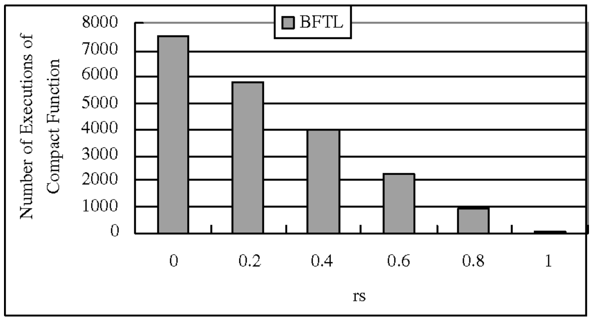
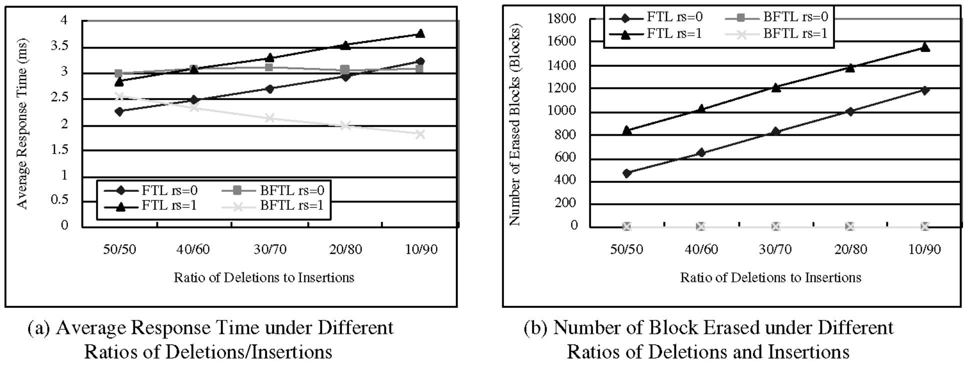


Fig. 6. Number of executions of compact function.

For the garbage collection issue, in Figure 5d, we can see that BFTL certainly suppressed the garbage collection activities when compared with FTL. In all experiments of BFTL, garbage collection did not even start yet. As a result, a longer lifetime of flash memory could be promised by BFTL. We must point out that experiments on a 512 MB NAND flash memory would result in very similar results observed in the experiments, even though garbage collection might come later, compared to the current experiments with a 4 MB NAND flash memory. Figure 6 shows the overhead introduced by the compaction of the node translation table. In Figure 6, we can see that the number of executions of compacting was reduced when the values of the inserted keys were in



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

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

Fig. 7. Experimental results of B-tree index structures maintenance.

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.

6.3 Performance of B-Tree Index Structures Maintenance

In this part of the experiment, we measured the performance of BFTL and FTL to maintain B-tree index structures. Under the workloads adopted in this part of experiment, records were 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 30% of total operations are deletions and the other 70% of total operations are insertions. For each run, 30,000 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 $rs = 0$ (see Section 6.2 for the definition of rs) were used as two representative experiment settings.

The x axes of Figures 7a and 7b denote the ratios of deletion/insertion. Figure 7a shows the average response time under different ratios of deletion/insertion. The average response time shows that FTL outperformed BFTL when the ratio of deletions/insertions changed from 50/50 to 20/80 with $rs = 0$ (the keys were randomly generated). Since the inserted keys were already randomly generated and then randomly deleted, more index units were chained in the node-translation table so that the visiting of a B-tree node was not very efficient. When the ratio of deletions/insertions was beyond 20/80 with $rs = 0$, BFTL outperformed FTL because the number of insertions was increased so that FTL could incur more pages written. With $rs = 1$, when the ratio of deletions/insertions changed from 50/50 to 10/90, the performance of BFTL greatly improved. The reason was that BFTL could have less pages written, when the values of the inserted keys were in an ascending order. As a result, when the ratio of deletion/insertion changed from 50/50 to 10/90 with $rs = 1$, BFTL could have better performance than FTL. Figure 7b shows the number of block erased

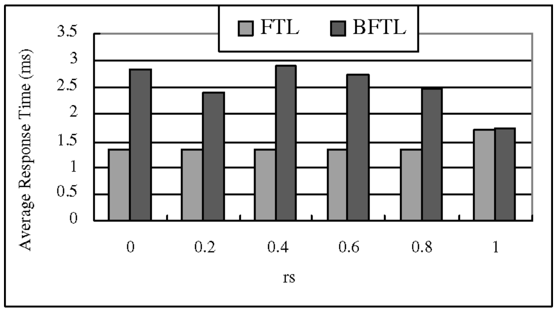


Fig. 8. Experimental results of B-tree index structures search.

in the experiments. The garbage collection activities were substantially reduced by BFTL; they had even not started yet in all experiments of BFTL.

6.4 Performance of B-Tree Index Structures Search

In this part of the experiment, we measured the performance of FTL and BFTL in the search of B-tree index structures. B-tree index structures were created by inserting 30,000 records under different values for *rs*. For each run of experiments, 3000 searches were randomly issued for different keys and the average response time of the searches was measured, as shown in Figure 8. Since the index units of B-tree nodes could be scattered over flash memory because of the commit policy, BFTL would need to read more sectors for the construction of B-tree nodes so that BFTL had longer response time in search operations than FTL did. The lengths of lists in the node-translation table had to be bounded to avoid lengthy searching time over BFTL. In the experiments, the worst case of the average response time of the searches over BFTL would be no more than four times that of FTL, because the length bound of the lists was set as four in the experiments. However, the experimental results in Figure 8 show that the average response time of the searches over BFTL was no more than twice that of FTL. In the meantime, BFTL did effectively reduce the number of pages writes and the number of erased blocks for applications with a lot of small updates over B-tree index structures.

6.5 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 into 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 benefits from buffering/caching records, however, it damages the reliability of BFTL because of power-failures. Reservation buffers with different size were evaluated to find a reasonably good setting. The experiment setups in Section 6.2 were used in this part of the experiment, but the value of *rs* was fixed at 0.5. The size of the reservation buffer was set between 10

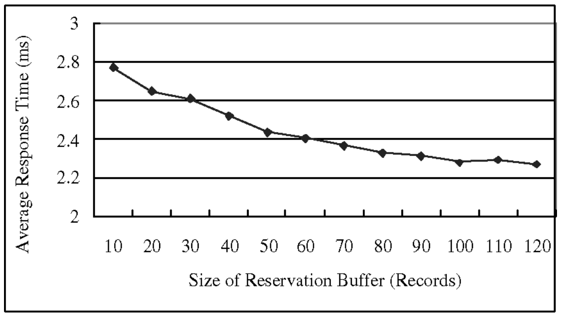


Fig. 9. Experimental results of BFTL under different sizes of the reservation buffer.

Table II. Energy Dissipations of BFTL and FTL (*joule*)

Creation		
	BFTL	FTL
$rs = 0$	27.09	28.33
$rs = 1$	14.79	32.65

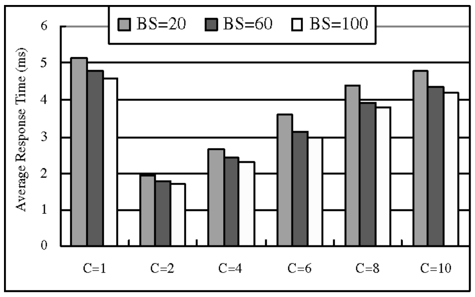
Maintenance		
	BFTL	FTL
50/50, $rs = 0$	25.28	18.52
50/50, $rs = 1$	21.56	23.24
10/90, $rs = 0$	26.15	26.25
10/90, $rs = 1$	15.53	30.47

and 120 records; the size was incremented by 10 records. Figure 9 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 to 60 records. After that, the average response time was linearly reduced and no significant improvement was observed. Since further increasing the size of the reservation buffer can damage the reliability of BFTL, the recommended size of the reservation buffer for the experiments was 60 records.

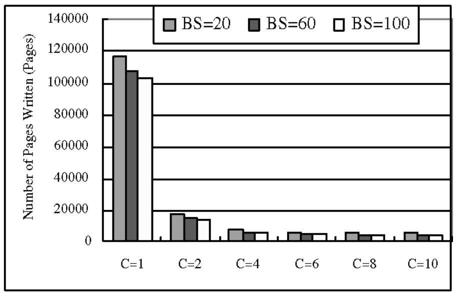
Energy consumption is also a critical issue for portable devices. According to the numbers of reads/ writes/ erases generated in the experiments, we calculated the energy consumption contributed by BFTL and FTL. The energy consumptions of reads/ writes/ erases are included in Table I. The calculated energy consumption of the experiments are listed in Table II. 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 efficiently 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.

6.6 Performance for Different Bounds

In this part of the experiment, we evaluated the performance of BFTL under different bounds of the lengths of lists (referred to as C hereafter) in the node translation that was 1, 2, 4, 6, 8, and 10. For each run of experiments, we



(a) Average Response Time under Different Bounds



(b) Number of Pages Written under Different Bounds

Fig. 10. Average response time and number of pages written under different bounds.

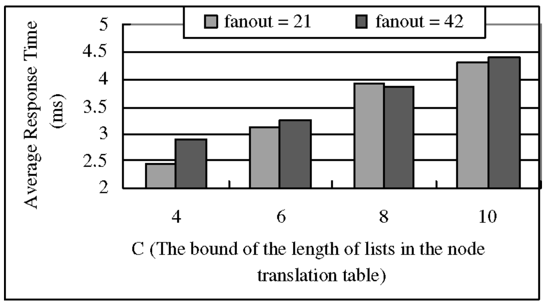


Fig. 11. Average response time under different fanouts.

inserted 30,000 records, *rs* was set to 0.5, and *BS* was among 20, 60, and 100. Figure 10a shows that the response time of record insertions under different *C*. We can see that when *C* was set to 1, the average response time was the worst, because the number of executions of compact function was increased dramatically. We could observe that when *C* was set to 2, the average response time showed the best performance. However, as shown in Figure 10b, when *C* was set to four, the number of pages written was reduced significantly compared to $C = 2$. Furthermore, when *C* was larger than four, no significant improvement were observed. As a result, for a longer lifetime of flash memory and reasonable response time, *C* was set to four in the experiments.

6.7 Performance for Different Fanouts

In this part of the experiment, we evaluated the performance of BFTL under different fanouts of a B-tree node. In the previous experiments, the fanout was set to 21, because the size of a B-tree node could fit into a sector. In this experiment, the fanout was 21 and 42, which would occupy one sector and two sectors for a B-tree node, respectively. For reflecting the variation of size of a B-tree node, we also varied the bound of the lengths of lists (referred to as *C* hereafter) in the node translation that was 4, 6, 8, and 10. For each run of experiments, we inserted 30,000 records and *rs* was set to 0.5. Figure 11 shows that the response time of record insertions under different fanouts and *C*. We can see that when C was set to four, the average response time with fanout = 42 was longer than that with fanout = 21. Because the bigger fanout would result in the larger size of a B-tree node, the larger size of a B-tree node soon was beyond C. The number of executions of compact function then increased accordingly. On the other hand, when C was increased to 6, 8, and 10, the average response time also increased, because the larger C alleviated the number of executions of compact function, but it also increased the number of pages read for constructing a B-tree node. We also can see that the larger fanout did not show significant improvement in Figure 10. Although the larger fanout provided the reduced height of a B-tree index structure, it also increased the size of a B-tree node, which resulted in the more pages read, and increased the number of executions of compact function under BFTL. As a result, we set the fanout to 21 and C to 4 in the previous experiments for obtaining a reasonable performance.

7. CONCLUSION

Flash-memory storage systems are very suitable for embedded systems such as portable devices and consumer electronics. As a result of 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 amount 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 future work. With the advance of flash-memory technology, large-scaled flash-memory storage systems may 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 overhead in flash-memory management could introduce a serious performance in system designs.

REFERENCES

Bayer, R. and McCreight, E. M. 1972. Organization and maintenance of large ordered indices. *Acta Informatica 1*, 173–189.

Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. 1996. An asymptotically optimal multiversion B-tree. *VLDB Journal 5*, 4, 264–275.

Chang, L. P. and Kuo, T. W. 2001. A dynamic-voltage-adjustment mechanism in reducing the power consumption of flash memory for portable devices. In *Conference on Consumer Electronic (ICCE)*. IEEE, LA.

Chang, L. P. and Kuo, T. W. 2002. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. San Jose, CA. IEEE, Washington, D.C.

Chang, L. P., Kuo, T. W., and Lo, S.-W. 2004. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems 3*, 4.

COMER, D. 1979. The ubiquitous b-tree. *ACM Computing Surveys 11*, 2, 121–137.

DOUGLIS, F., CACERES, R., F. KAASHOEK, K., LI, B. M., AND TAUBER, J. A. 1994. Storage alternatives for mobile computers. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX. 25–37.

FREESTON, M. 1995. A general solution of the n-dimensional b-tree problem. In *SIGMOD Conference*. San Jose, CA. ACM, New York.

Ftl logger exchanging data with ftl systems.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability*. Freeman, San Francisco, CA.

HAN-JOON, K. AND SANG-GOO, L. 1999. A new flash memory management for flash storage system. In *Proceedings of the Computer Software and Applications Conference (COMPSAC)*. IEEE, Washington, D.C.

http://www.linuxgazette.com/issue55/florido.html.

http://www.samsung.com/products/semiconductor/flash/index.htm.

KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. 1995. A flash-memory based file system. In *USENIX Technical Conference on Unix and Advanced Computing Systems*.

KIM, H. J. AND LEE, S. G. 1999. A new flash memory management for flash storage system. In *Annual International Computer Software and Applications Conference*. Phoenix, AZ. IEEE.

KIM, J., KIM, J. M., NOH, S. H., MIN, S. L., AND CHO, Y. 2002. A space-efficient flash translation layer for compact-flash systems. *IEEE Transactions on Consumer Electronics 48*, 2 (May).

KUO, T. W., WEY, J. H., AND LAM, K. Y. 1999. Real-time data access control on b-tree index structures. In *International Conference on Data Engineering (ICDE)*. Sydney, IEEE, Washington, D.C.

Lfs file manager software: Lfm.

1998. *compact flash*^{TM} 1.4 specification.

1999. *smartmedia*^{TM} specification.

PARK, C., KANG, J., PARK, S. Y., AND KIM, J. 2004. Energy-aware demand paging on nand flash-based embedded storages. In *International Symposium on Low Power Electronics and Design (ISLPED)*.

PARK, C., SEO, J., BAE, S., KIM, H., KIM, S., AND KIM, B. 2003. A low-cost memory architecture with nand xip for mobile embedded systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. IEEE/ACM/IFIP, CA.

RAMAKRISHNAN AND GEHRKE. 2003. *Database Management Systems*. McGraw-Hill, New York.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (Feb.), 26–52.

Software concerns of implementing a resident flash disk.

Understanding the flash translation layer(ftl) specification.

VAZIRANI, V. V. 2001. *Approximation Algorithm*. Springer, New York.

WOODHOUSE, D. Jffs: The journaling flash file system.

WU, C. H., KUO, T. W., AND YANG, C. L. 2004. Energy-efficient flash-memory storage systems with interrupt-emulation mechanism. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. IEEE/ACM/IFIP, Stockholm, Sweden.

WU, C. H., KUO, T. W., AND CHANG, L. P. 2006a. Efficient initialization and crash recovery for log-based file systems over flash memory. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*. Dijon, France. ACM, New York.

WU, C. H., KUO, T. W., AND YANG, C. L. 2006b. A space-efficient caching mechanism for flash-memory address translation. In *International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. Gyeongju, Korea. IEEE, Washington, D.C.

WU, M. AND ZWAENEPOEL, W. 1994. envy: A non-volatile, main memory storage system. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York.

Yet another flash filing system.

YOKOTA, H., KANEMASA, Y., AND MIYAZAKI, J. 1999. Fat-btree: An update-conscious parallel directory structure. In *International Conference on Data Engineering (ICDE)*. 448–457. IEEE, Washington, D.C.

Received March 2003; revised July 2005 and February 2006; accepted May 2006