A Reliable B-Tree Implementation over Flash Memory

Xiaoyan Xiang, Lihua Yue, Zhanzhan Liu, Peng Wei
Department of Computer Science and Technology
University of Science and Technology of China, Hefei, P.R.China
{xyxiang, zzliu, pwei}@ustc.edu, llyue@ustc.edu.cn

ABSTRACT
Flash memory has been widely used in various embedded computing systems and portable devices in recent years because of its small size, shock-resistance, low-power consumption and non-volatile properties. To hide the disadvantages of flash memory such as out-of-place update, a flash translation layer (FTL) is usually used for providing transparent block-device emulation. But when index structures are implemented over FTL, intensive overwrite operations caused by record inserting, deleting, modifying and index reorganizing could not only degrade the performance significantly but also reduce the life of flash memory. To address the problem, BFTL and IBSF are proposed. However, neither of them could avoid the loss of records and incompatibilities when system crash occurs. In this paper, a reliable B-tree implementation called RBFTL is presented for flash-memory storage systems. It is placed between the application layer and FTL. RBFTL could minimize the loss of data and eliminate incompatibilities effectively and efficiently when system crashes. The experimental results also show that RBFTL yields a better performance than FTL.*

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.

General Terms
Reliability, Algorithm, Design.

Keywords
Flash Memory, FTL, IBSF, B-Tree, Crash Recovery.

1. INTRODUCTION
In recent years, flash memory has been widely used in various embedded computing systems and portable devices such as PDAs (personal digital assistants), HPCs (handheld PCs), MP3 players and mobile phones because of its small size, shock-resistance, low-power consumption and non-volatile properties [4, 8].

There are mainly two types of flash memories: NOR and NAND. NOR flash memory usually consists of a number of blocks and each block has a fixed number of words. The typical block size and word size is 64KB and 2B, respectively. The read/write operations of NOR flash can be performed either on a word basis or on a byte basis, while the erase operation is performed on a block basis. While on word basis, each cost of the read/write operations is about 70ns and 9μs, respectively. The cost of erase operation is about 0.7s [7]. NAND flash memory usually consists of many blocks and each block is of a fixed number of pages. The typical block size and page size is 16KB and 512B, respectively. The read/write operations of NAND flash are performed on a page basis, while the erase operation is performed on a block basis. Each cost of the read/write/erase operations is about 15μs, 200μs and 2ms, respectively. NAND flash has three main characteristics: out-of-place update, asymmetric read/write/erase speed and limited block erase count. In order to hide the limitation of out-of-place update, an intermediate software layer called a flash translation layer (FTL) [3] is usually used to provide transparent block-device emulation. On the other hand, the write/erase operations are relatively slow compared with the read operation, and write operations could introduce erase operations. Therefore, the number of write operations should be minimized.

The capacity of a NAND flash chip has reached 32Gb in year 2006 [9], and it is still increasing rapidly. Flash-memory storage systems have become good mass storage solutions, thus index structures are more and more needed to upgrade the access performance in large amount of data. A B-Tree [5, 6] consists of a hierarchical structure of data and it provides very efficient operations to find, insert and delete the data in the large storage devices. There are two kinds of nodes in a B-Tree: internal nodes and leaf nodes. Usually, a B-tree node can be stored to a page in NAND flash memory. Although FTL could provide an efficient mapping algorithm from the logical pages number to the physical location to reduce write/erase operations, when B-tree structures are implemented on FTL, its performance still degrades significantly. Because every record insertion/deletion/modification operation needs to update one or more B-tree nodes, and frequent insertion/deletion/modification operations of records could generate large numbers of overwrite operations. Moreover, so many write operations may introduce quantities of garbage collection operations which could not only degrade the overall performance but also decrease the life of flash memory.

To address the problem, BFTL [1] and IBSF [2] were proposed for efficiently implementing B-tree structures on NAND flash memory. Both of them are implemented over FTL and handle intensive byte-wise overwrite operations due to B-tree access. BFTL and IBSF could significantly reduce the number of write operations.

* This research was supported by the National High Technology Research and Development Program of China (2006AA1327).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. SAC’08, March 16-20, 2008, Fortaleza, Ceará, Brazil. Copyright 2008 ACM 978-1-59593-753-7/08/0003…$5.00.
operations and improve the overall performance. However, both BFTL and IBSF have disadvantages when system crash occurs (e.g., power-off). First, records buffered by BFTL and IBSF may have not been written to flash memory when system crash occurs. This means the user’s data may lose. Second, neither BFTL nor IBSF could ensure that a record and its corresponding index units are all written to flash memory or none. Thus incompatibilities may exist and lead to mistakes. System crash may also bring incompatibilities if we directly implement B-tree index structures over FTL. But none of FTL, BFTL or IBSF provides an effective mechanism to detect and handle incompatibilities. The log mechanism in traditional journaling file systems is helpful to handle this problem, but it cannot be directly employed on NAND flash memory because frequent log-writing operations could not only degrade the overall performance but also decrease the life of flash memory due to the special properties of NAND flash.

In this paper, we propose a reliable B-tree implementation, called RBFTL, for flash memory storage systems. It sits between the application layer and FTL. RBFTL writes records in the buffer to NAND flash memory in time, so it could minimize the loss of data. Furthermore, RBFTL backs up the index units into NOR flash memories before writing records. Thus, if system crash occurs, it could detect and eliminate incompatibilities effectively and efficiently. We must point that although our work is based on FTL, the idea of this paper could be easily extended to a native flash-memory file system.

2. RELATED WORK

IBSF is a buffer management schema for implementing a B-tree on NAND flash memory. As shown in Figure 1, IBSF is implemented over FTL and it has a buffer in RAM to temporarily hold the newly generated records (referred as “dirty records” for the rest of this paper) and index units. The usage of index units is to reflect primary-key insertions and deletions to the B-Tree nodes caused by the dirty records.

When a record is inserted, deleted or modified, one or more B-tree nodes may be modified and one or more index units may be constructed. For example, as shown in Figure 1, when 7 is inserted to B-tree after 6, five index units which are [5, C, d], [5, A, i], [6, C, d], [6, D, i] and [7, D, i] are generated, and [7, D, i] means that primary-key value 7 should be inserted to B-tree node D. Index units are held in the index buffer according to the following insertion policy and deletion policy. If the new index unit is insert-type and an index unit with the same type and primary key value already exists in the index buffer, IBSF updates it with the new index unit. If the new index unit is of a delete-type and the index buffer keeps a redundant index unit which is insert-type and has the same primary key value, IBSF eliminates the redundant index unit and inserts the new index unit into the index buffer. If the new index unit is of a delete-type and the index buffer has another index unit with the same type and node identifier, IBSF updates the flag of the new index unit in the existing index unit. Otherwise, IBSF directly inserts the new index unit to the index buffer. For example, when [6, C, d] is inserted, [6, C, i] is eliminated and [6, C, d] updates [5, C, d], according to the deletion policy. When the index buffer is filled, IBSF has to commit index units from the index buffer to NAND flash memory. IBSF chooses the first index unit in the index buffer, and then collects the index units both in the index buffer and in flash memory which have been related to the first index unit. Then, the redundant index units are eliminated and other index units are committed into one page. The above process is repeated until all index units in the index buffer are written to flash memory. Different to index unit, the size of a record is large, so the dirty records do not need to be packed and they are directly written to flash memory.

As shown in Figure 1, if B-tree is directly adopted over FTL, up to six write operations may be needed to handle the modification of B-tree nodes when we insert 6, 7, 8 and 9 because a split occurs while inserting 7. But IBSF only needs three write operations because only node A, C and D are modified.

However, IBSF has two drawbacks as follows: First, when system crash occurs (e.g., power-off), some records in the buffer may have not been written to flash memory. It means that the user’s data may lose, and this is not considered by BFTL either. Second, IBSF could not ensure that a record and its corresponding index units are all written to flash memory or none. Thus when system crash occurs, incompatibilities which could lead to mistakes may exist. For example, if [9, D, i] is written to flash memory while the corresponding record is not successfully written, user who searches the record with 9 as primary key value may get wrong data. The B-tree may be in an invalid state if only certain of index units are written to flash memory. System crash may also bring incompatibilities if B-tree indices are directly implemented over FTL. But none of FTL, BFTL or IBSF provides an effective mechanism to detect and handle incompatibilities.

3. THE IMPLEMENTATION OF RBFTL

3.1 Overview

We have reviewed IBSF and discussed the drawbacks in section 2. In this section, a reliable B-tree implementation, called RBFTL, is presented for flash-memory storage systems. As shown in Figure 2, RBFTL is placed between the application layer and FTL. It is responsible for processing the B-tree related requests from upper applications and sending block-device requests to FTL. RBFTL consists of index buffer, insertion/deletion/commit policies, logger, crash recovery unit and two NOR flash memories. The capacity of each NOR flash memory is very small compared with the NAND flash memory. The index buffer keeps a fixed number of index units.
units and the NOR flash memories backup index units before they are written to NAND flash memory. The logger constructs index units, and writes the index units and dirty records to NOR/NAND flash memories. The insertion/deletion/commit policies provide how to deal with the dirty records and index units. The crash recovery unit is responsible for detecting and handling incompatibilities when system crash occurs. Because the erase speed of NOR flash is very slow, we use two NOR flash memories in turn. When one NOR flash memory is being used to backup index units, the other one can be erased synchronously. Thus RBFTL does not need to wait for garbage collection which costs so much.

The architecture of RBFTL is shown in Figure 3. When a record is inserted, deleted, or modified by upper applications, the logger of RBFTL generates one or more index units to reflect primary-key insertions and deletions to the B-Tree nodes. These index units are inserted to index buffer and NOR flash memories by the logger according to the insertion/deletion policies. After the related index units are written to NOR flash memories, the dirty record is written to NAND flash memory or deleted from NAND flash memory through FTL. When the index buffer is filled or does not have enough space to accept the new index units, a commit operation should be performed. All the index units in the index buffer are packed and then written to NAND flash memory.

RBFTL sets signs in NOR flash memories to denote whether the commit operation is performed successfully.

In the following sub-sections, we shall present the detail of the insertion/deletion policies, the commit policy and the crash recovery process.

### 3.2 Insertion and Deletion Policies

In RBFTL, when applications insert, delete, or modify a record, one or more index units are generated by the logger to reflect primary-key insertions and deletions to the B-Tree nodes. Accordingly, an index unit can be of insertion-type or deletion-type. The processing is done as follows: First, if there is no enough space in the index buffer to accept the newly generated index units, the logger generates a commit operation to vacate the index buffer. We always commit the index units related to one record at one time, and this could reduce incompatibilities. Otherwise, the logger first backups these index units in a NOR flash memory. Then the index units are inserted to index buffer according to the same insertion and deletion policies as adopted in IBSF. Second, after the backup of the related index units is completed, a record-insert sign `recins` is set in NOR flash memory to denote that a dirty record is about to be written to NAND flash memory, or a record-delete sign `recdel` is set to denote that a record is about to be deleted. Then the record is written to NAND flash memory or deleted from NAND flash memory through FTL. Finally, a completion sign `reccompl` is set to denote that the above steps are performed successfully. We define the index units between two adjacent completion signs in NOR flash memory as an index-unit-portion. All the index units in an index-unit-portion are related to the same record.

When a record insertion operation occurs, if more than one index units are generated, the logger always puts the index unit which contains the logic address of the new record on the last position when it backups the index units. For example, as shown in Figure 3, five index units are generated when a record with 7 as primary key value is inserted, and we last backup [7, D, i] because it contains the logic address of the record (not visible in the figure). This could help the crash recovery process. On the other hand, when a record deletion operation occurs, the logger always puts the index unit which contains the logic address of the record on the first position because it is usually generated first. It is worth mentioning that the records still stay in RAM no matter they are written to NAND flash memory or not, until a commit operation is performed. By doing this, if a recent record is requested, RBFTL could directly get it from RAM and does not need to get it from flash memory through the B-tree.

### 3.3 Commit Policy

When the index buffer is filled or does not have enough space to accept the newly generated index units, a commit operation should be performed to vacate the index buffer. First, a commit-start sign `comstr` is set in NOR flash memory to denote that a commit operation is about to be performed. Then all the index units in index buffer are packed in the manner employed in IBSF, and then they are written to NAND flash memory through FTL. Finally, the logger sets a commit-complete sign `comcomp` in NOR flash memory to denote that the commit operation is performed successfully.
We define the data between two adjacent commit-complete signs in NOR flash memory as a log-segment. A log-segment consists of some index-unit-portions, a commit-start sign and a commit-complete sign. If there is no enough space in the current active NOR flash memory to store a log-segment, it is marked as inactive and the other NOR flash memory is used to store the rest data. The inactive memory should be erased to obtain free space after the log-segment is completely stored. This can ensure the wear-leveling property in NOR flash memories.

3.4 Crash Recovery

When system crash occurs, some records might lose and incompatibilities might exist. By writing records to NAND flash memory in time, RBFTL could minimize the loss of data. In this sub-section, a crash recovery process is proposed to detect and eliminate incompatibilities when system is rebooted after crash. The process is controlled by a crash recovery algorithm.

First, RBFTL searches the latest log-segment (might not complete) in the active NOR flash memory. Use $W$ to represent the last word in the log-segment. The proposed crash recovery algorithm is shown as follows:

1. If $W$ is comcomp, return.
2. If $W$ is constr or reccomp, the following steps are performed:
   a) Read the index-unit-portions one by one from the beginning of the log-segment to the end;
   b) For each index-unit-portion, insert its index units into the index buffer according to the insertion/deletion policies mentioned above. But no data needs to be written to NOR flash memory and no record needs to be written to NAND flash memory or deleted from NAND flash memory;
   c) Write all the index units in the index buffer to NAND flash memory according to the commit policy mentioned above. But no data needs to be written to NOR flash memory;
   d) Set a commit-complete sign comcomp behind the log-segment.
3. If $W$ is recins, the following steps are performed:
   a) Read the index unit ahead of $W$ and get the logic address of its related record, then delete the record through FTL or take other process according to the requirement of applications;
   b) Skip the index-unit-portion which contains $W$. For other index-unit-portions in the log-segment, take the same process as step 2 above.
4. If $W$ is recdel, the following steps are performed:
   a) Read the index unit ahead of $W$ and get the logic address of its related record, then delete the record through FTL;
   b) For all the index-unit-portions in the log-segment, take the same process as step 2 above.
5. If $W$ is not any sign above, skip the index-unit-portion which contains $W$. For other index-unit-portions in the log-segment, take the same process as step 2 above.

If $W$ is comcomp, it means that the last commit operation has been performed successfully and no record has been written to NAND flash memory after the commit operation. Hence no incompatibility exists and nothing is needed to do.

If $W$ is constr or reccomp, it means that the last commit operation is aborted or has not been performed. If the commit operation is aborted, the index units of some B-tree nodes might be written to NAND flash memory while the others are not. Thus when system is rebooted, some of the B-tree nodes which need update might be updated successfully while the others are not. If the commit operation is not performed, no B-tree node is updated. But the related records have been written to NAND flash memory. To eliminate these incompatibilities, we need to redo the process mentioned in sub-section 3.2 and 3.3 to deal with the index units. For the nodes which have been updated, doing the process again should have no influence. For the nodes which have not been updated, doing the process could update them. After eliminating the incompatibilities, a commit-complete sign is set to denote the completion of recovery.

If $W$ is recins, it means that a record insertion/deletion operation is aborted and the commit operation has not been performed. So we first deal with the record and skip the aborted operation, then redo the process in step 2 for other index-unit-portions to update the related B-tree nodes. If $W$ is recdel, similar steps are performed.

If $W$ is not any sign above, it means that an index unit has not been backed up successfully and the related record has not been written to NAND flash memory. Therefore, we only need to skip the incomplete index-unit-portion and redo the process in step 2 for other index-unit-portions.

The above analysis shows that the proposed crash recovery algorithm can effectively detect and eliminate the incompatibilities caused by system crashes. The crash recovery process only needs to read one log-segment from NOR flash memory and write a few data to NAND/NOR flash memories, so it is very efficient.

4. EXPERIMENTS

To evaluate the performance of RBFTL, a series of experiments had been done. In the experiments, the capacity of NAND flash memory was 4MB while the capacity of each NOR flash memory was 1MB. Since the contents of records were not stored, a 4MB NAND flash memory was sufficiently large for the experiments. The index buffer was configured to hold 50 index units in the experiments. A B-Tree was built over FTL to evaluate the performance of FTL. The size of each B-Tree node was no more than 512B and the fan-out was 25. We inserted 24,000 records to the B-tree for each experiment. It is worth mentioning that RBFTL cached a few B-Tree nodes which were frequently accessed in RAM.

In the experiments, we measured the number of pages read, written and erased. The number of index units written to NOR flash memories and the total consumed time were also measured. A ratio RS was used to denote the value distribution of the inserted keys. If RS equals to 0, all the keys were randomly generated. If RS equals 1, that means the value of the inserted keys were in an ascending order. Figure 4 shows the experimental results. Figure 4.(a) shows the number of page write operations executed by FTL and RBFTL, and it also shows the number of index units written to NOR flash memories by RBFTL. Though the write speed of NOR flash is lower than NAND flash, writing an index unit is still faster than writing a page because the size of
an index unit (16B in our experiments) is very small. Figure 4.(c) shows that RBFTL reduced the number of erase operations significantly. This can help to prolong the life of flash memory. As shown in Figure 4.(d), the total time consumed by RBFTL was less than FTL, regardless the value of RS. The experimental results show that RBFTL yields a better performance than FTL. With the write speed of NOR flash increasing, the performance of RBFTL should be better. We also measured the time cost by crash recovery and the results show that the time is of the order of 10ms, regardless the capacity of NAND flash memory.

5. CONCLUSION
In this paper, a reliable B-tree implementation called RBFTL is presented to reliably implement B-tree indices over flash memory. RBFTL writes dirty records to NAND flash memory in time, thus it could minimize the loss of data. Furthermore, RBFTL backups the index units in NOR flash memories before writing records. If system crash occurs, RBFTL could detect and eliminate incompatibilities effectively and efficiently according to the crash recovery algorithm. Therefore, RBFTL could overcome the disadvantages of FTL, BFTL and IBSF. Finally, the experimental results also show that RBFTL yields a better performance than FTL.

With the rapid development of flash memory technology, flash memory may be not only widely employed in embedded systems and portable devices but also potential to replace magnetic disks in large-scale data storage field. How to manage data records and index structures over huge flash memory should be further exploited in the future.

6. REFERENCES

Figure 4. Experimental results.