AN EFFICIENT RECOVERY SCHEME FOR BUFFER-BASED B-TREE INDEXES ON FLASH MEMORY

VanPhi Ho, Seung-Joo Jeong, Dong-Joo Park

School of Computer Science and Engineering,
Soongsil University
Seoul, Korea
{hvphi, qkaxhf1007, djpark }@ssu.ac.kr

ABSTRACT

Recently, flash memory has been widely used because of its advantages such as fast access speed, nonvolatile, low power consumption. However, erase-before-write characteristic causes the B-tree implementation on flash memory to be inefficient because it generates many flash operations. To address this problem, variants of buffer-based B-tree index have been proposed for flash memory which can reduce a number of write operations. Since these B-trees use a main-memory resident index buffer to temporarily store newly created index units, their data may be lost if a system crash occurs. This study introduces a novel recovery scheme for the buffer-based B-tree indexes on flash memory, called ERS. ERS can minimize the risk of losing data by deploying logging and recovery policies. The experimental results show that ERS yields a good performance and helps the buffer-based B-tree indexes improve the reliability.

KEYWORDS

B-tree index, flash-aware index, flash memory.

1. INTRODUCTION

Flash memory [1-2] has been widely used because it has many positive features such as high-speed access, low power consumption, small size and high reliability. Besides these advantages, it has some downsides including erase-before-write, limited life cycle. In order to access data on the flash memory as accessing hard disk drives, hosts need to use a firmware module named Flash Translation Layer (FTL) [1]. FTL translates the logical address to physical address between the host and flash memory. The FTL has two main features: address mapping and garbage collection. Many FTL algorithms have been used to confine the limitation of physical characteristics and enhance the performance of flash memory.

By using FTL algorithms, the performance of a flash memory has been improved. However, implementing B-tree index directly on flash memory may not be efficient because the erase-before-write characteristic. Updating B-tree nodes causes the overwrite operations on flash memory occur frequently. To address these problems, variants of B-tree index have been proposed for flash memory. Among these B-tree variants, there are some B-trees using a main-memory resident index buffer (called buffer-based B-tree for short) to temporarily store newly created index units in order to reduce the number of flash operations. However, using the main-memory resident index buffer causes the index data in the buffer may be lost when a sudden power-off occurs.

Natarajan Meghanathan et al. (Eds) : NETCOM, NCS, WiMoNe, GRAPH-HOC, SPM, CSEIT - 2016
pp. 179– 186, 2016. © CS & IT-CSCP 2016 DOI : 10.5121/csit.2016.61515

This paper present a new recovery method for the buffer-based B-tree indexes on Flash Memory, called ERS. Whenever the index units are inserted into the index buffer, the ERS backs up the index units to an area called logger which is located in flash memory to avoid losing its data. If the system reboots after a crash, the backed up index units in the logger are read back to the index buffer. ERS could minimize the loss of data.

The experimental results indicate that our proposed scheme achieves a good performance and it helps the buffer-based B-tree indexes improve the reliability.

The paper is organized as follows: Section 2 reviews background and related works. The design of ERS and its operations are presented in Section 3. Section 4 experimentally evaluates the efficiency of ERS, and finally, Section 5 concludes the paper.

2. BACKGROUND AND RELATED WORKS

Flash memory is a storage device whose data is nonvolatile. It is widely used nowadays because its strong points. Different from a traditional hard disk drive, flash memory is consisted of a number of NAND flash memory arrays, a controller, and an SRAM. NAND flash memory arrays are organized in many blocks. Each block contains a fixed number of pages (e.g. 32, 64). A page is the smallest unit of read and write operations while the block is the smallest unit of erase operations. Similar to the hard disk drive, flash memory supports all basic operations: read, write and erase. The read operation is the fastest one, that is about 10 times faster than a write operation. The erase operation is very time-consuming, which takes about 2ms. The erase operation is over 10 times slower than a write operation. Also, as mentioned above, the main drawback characteristic of NAND flash memory is that it has erase-before-write architecture. Moreover, the life cycle of flash memory is limited. The number of erase cycles for a block is bounded about 100,000 times. Therefore, frequent erasing of some particular locations may deteriorate both the overall performance and lifetime of the flash memory. Since flash memory owns these physical characteristics, it requires an intermediate module called Flash Translation Layer (FTL) for translating the address mapping, managing and controlling data. By using FTL, the general performance of flash memory is improved and quickly deploy disk-based applications without any modifications.

B-tree index [3] is a data structure which is popularly used in many file systems and database management systems because of quickly access capability. However, implementing B-tree directly on flash memory may suffer from degradation of the efficiency of B-tree index as well as the lifecycle of flash memory because of the erase-before-write limitation of flash memory.

To address these problems, variants of B-tree have been proposed for flash memory. Wu et al. presented BFTL [4], the first B-tree variant. BFTL is consist of a node translation table and a reservation buffer. Every newly created index unit which reflects the inserted, deleted or modified records is temporarily stored in the reservation buffer. When the reservation buffer is full, all index units in the buffer are flushed to flash memory in FIFO order by an internal operation of BFTL called commit. Since some index units for the same node may be written in various pages, a node translation table is used. The node translation table collects all index units and maintain the information of the pages having the index units of the same B-tree node. As a result, BFTL reduces the number of flash operations. However, many read operations is needed to access a B-tree node because the index units of one node may be scattered on many different flash pages. Moreover, since the buffer is a volatile storage, its data may be lost when a sudden power-off occurs leading to the stored B-tree in the flash memory may be an unstable structure. And then, it may yield serious problems when managing a number of data.

In order to solve the drawbacks of BFTL, a new index buffer management scheme named IBSF [5] was proposed. The main idea of IBSF is to store all index units associated with a B-tree node onto one page, so IBSF does not need the node translation table. Similar to BFTL, IBSF temporarily stores newly created index units into the index buffer. When flushing records from the index buffer to flash memory, IBSF selects victim index units by identifying the records to be inserted into the same logical node. This prevents them from spreading across several flash pages. Thus, IBSF reduces the search overhead of BFTL. However, due to the fact that there are a lot of index units still in the index buffer of IBSF, its data may be lost when a sudden power-off occurs similarly to BFTL.

Later on, a write-optimized B-tree layer for NAND Flash memory (WOBF) [6] was proposed. Basically, WOBF inherits the advantages of BFTL and IBSF. It employs the index buffer and the node translation table used in BFTL and the commit policy of IBSF. Its performance is improved by sorting all the index units in the index buffer before performing commit operations. Sorting all the index units prevents the index units belonging to the same node from being scattered over many pages. This reduces the number of read operations when building a logical node. Nevertheless, similar to BFTL and IBSF, WOBF still suffers from losing data when a sudden power-off occurs because the index buffer is volatile.

Summary, the above buffer-based B-tree indexes reduce the number of writes when building a B-tree by exploiting the main memory index buffer. However, since the index buffer is a volatile storage, their data may be lost when a sudden power-off occurs leading to the stored B-tree index in the flash memory may be an unstable structure. Therefore, it may yield serious problems when managing a number of data.

3. THE DESIGN AND IMPLEMENTATION OF ERS

3.1. The design of ERS

This section presents a novel recovery scheme for the buffer-based B-tree indexes on flash memory, called ERS, to efficiently restore the data of buffer-based B-trees when a sudden power failure occurs. Its objective is to significantly reduce the risk of losing data of buffer-based B-tree indexes when a B-tree is built. In order to achieve the aforementioned goal, we maintain a flash-memory resident logger which uses some blocks of flash memory to temporarily stores newly created index units whenever the newly created index units are inserted into the index buffer. Figure 1 shows the architecture of ERS comprising an index buffer, buffer-based B-tree policies, a logger and a recovery module.

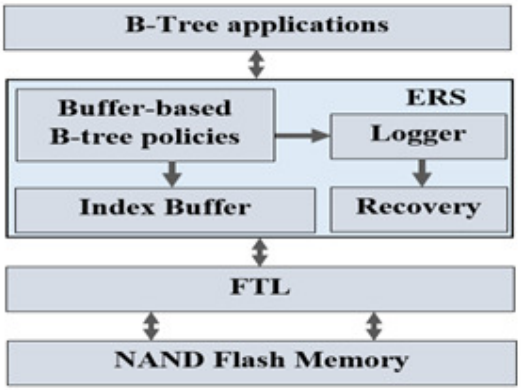


Figure 1. Overall architecture of ERS

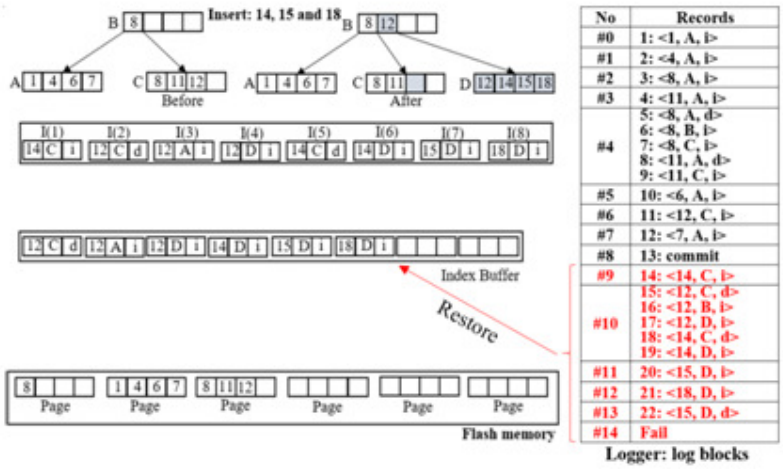
The buffer-based B-tree policies module basically manages the index unit in the index buffer according to the buffer-based B-tree algorithms. The logger is located in flash memory to avoid losing its data. It uses some blocks of flash memory to store all newly generated index units. These blocks are called log blocks. The logger adopts the logging mechanism [7] for the recovery so that it sequentially records the newly created index units whenever a B-tree node is modified. Since writing data sequentially into the logger, the overhead of this writing is relatively small [8-9]. When a commit operation is performed successfully (e.g. all the index units are written onto flash memory), a completion commit sign is set in the logger. The recovery is triggered when the system restarts after a crash. It detects and eliminates incompatibilities by restoring the state of the system just before the crash took place. By using the recovery module, ERS ensures the durability of all the data before the crash.

3.2. The implementation of ERS

3.2.1. Logging policy

For recovering data after the system crash, BMS writes the newly created index units to the logger sequentially. When a record is inserted into B-tree or deleted from B-tree, one or more index units are created to reflect the insertion/deletion. After inserting the newly created index units into the index buffer, ERS backups these index units to the logger simultaneously. The newly created index units are temporarily stored in the index buffer based on the policies of the buffer-based B-tree indexes. Owing to the limitation of the index buffer size, all index units in the index buffer are flushed to the flash memory when the index buffer is full by using the commit operation. When a commit operation is performed successfully, a commit record is created in the logger to denote that all data is written to flash memory. The commit record is a checkpoint that denotes a commit operation is finished successfully. Since the size of an index unit is much smaller than that of a flash page, all the index units related to one record are written to one page of the log blocks. This reduces the number of write operations onto the logger and saves a lot of space of the log blocks.

Figure 2 presents an example of the logging policy. Supposing that a buffer-based B-tree is built by inserting 10 records having key values as 1, 4, 8, 11, 12, 6, 7, 14, 15 and 18.



No	Records
#0	1: <1, A, I>
#1	2: <4, A, I>
#2	3: <8, A, I>
#3	4: <11, A, I>
#4	5: <8, A, d> 6: <8, B, I> 7: <8, C, I> 8: <11, A, d> 9: <11, C, I>
#5	10: <6, A, I>
#6	11: <12, C, I>
#7	12: <7, A, I>
#8	13: commit
#9	14: <14, C, I>
#10	15: <12, C, d> 16: <12, B, I> 17: <12, D, I> 18: <14, C, d> 19: <14, D, I>
#11	20: <15, D, I>
#12	21: <18, D, I>
#13	22: <15, D, d>
#14	Fail

Figure 2. Logging and recovery policy

To build the B-tree, the index units are created to reflect the insertions or deletions and then they are inserted into the index buffer. Simultaneously, these index units are written to the logger (e.g. page #0 to page #7 in the first block of log blocks). In this example, the index buffer is already

full when index unit <14, C, i> is created. At this time, all index units in the index buffer are written to flash memory and then a commit record is set in the logger (e.g. stored in page #8 of the first log block). After vacating the index buffer, the newly generating index units are inserted into the index buffer continuously according to the insertion and deletion policies. Additionally, since the size of the logger is limited, the logger eventually is fulfilled. Therefore, some log blocks in the logger should be erased timely to vacate space.

3.2.2. Recovery policy

For reliability and compatibility of buffer-based B-trees, ERS performs the recovery policy when the system is rebooted after the crash. The recovery policy is described as follows: First, ERS finds the last commit record in the logger because all index units are written into flash memory successfully before the commit record is created. Then it redoes operations by restoring all records after the last commit records in the logger to the index buffer. At this point, ERS operates normally by applying insertion, deletion and commit policies.

As shown in figure 2, supposing that the crash occurs after the record having key value 15 is deleted in node D (e.g. record 22 <15, D, d> in the figure). According to the recovery policy, ERS searches the last commit record in the logger. In this example, the last commit is the 13th record (page #8) in the logger. After getting the last commit record, ERS redoes operations by reading log records from 14th to 22th and inserts them into the index buffer. In this example, if there two more index units are inserted into the index buffer, a commit operation will be performed because the index buffer is full. So that, all the data that have not been written into flash memory is recovered successfully.

Processing recovery operation under this order allows ERS to reduce the number of write operations and garbage collections of the buffer-based B-trees. By using logging and recovery policies, ERS is much more reliable and compatible than the buffer-based B-trees. In addition, flash memory cards are sensitive to electrostatic discharge (ESD) damage, which can occur when electronic cards or components are handled improperly, results in complete or intermittent failures. Therefore, deploying ERS will be good in practical systems.

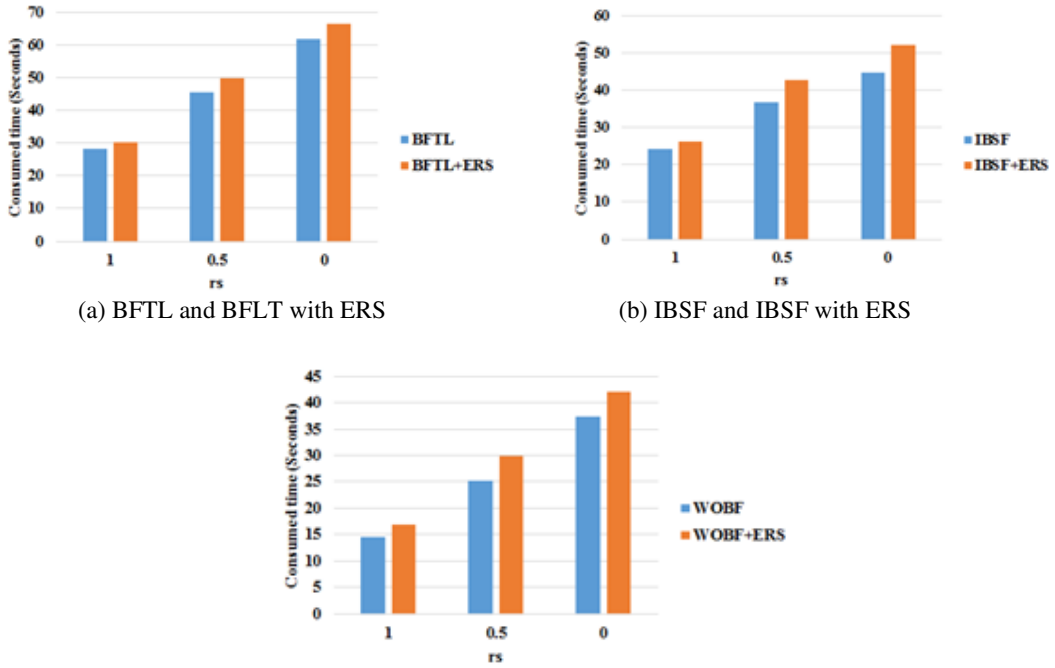
4. PERFORMANCE EVALUATION

This section shows the experimental results achieved by applying the proposed ERS and compares its performance to that of the buffer-based B-trees. All variant B-trees were performed on a NAND flash simulator which might be able to count the internal flash operations (read/write/erase). This simulator was configured for 64MB SLC NAND flash memory with 528 byte page size and 16 Kbyte block size. Every node of B-trees had 64 entries, each of which contained 4 byte integer key to search and a 4 byte pointer to point to the child node. The size index buffers are fixed as 64, and the index keys were unique integers in the range of 1 – 100,000. The performance of the buffer-based B-trees and ERS were assessed in terms of performance metrics: the average time to build B-trees and the recovery time. In order to control the key value distribution, a ratio called rs (ratio of the key sequence) was used. If the ratio was equal to 1, the key values were in ascending order. However, if the rs was equal to 0, the key values were randomly generated.

4.1. Performance of the B-trees creation

In this section, we assess the performance of ERS based on time consumption when building B-trees. Figure 3 presents the consumed time when constructing the buffer-based B-trees by inserting 100000 records. Overall, ERS yields about 8.2-11.3% overheads compared to those of the buffer-based B-trees on average. Concretely, ERS yields 8.19% overheads compared to that of BFTL, 10.56% overheads compared to that of IBSF and 11.32% overheads compared to that of

WOBF. The reason for these overheads is that ERS writes and manages the log record to the logger whenever a B-tree node is updated. However, the gap of their performance is smaller than we expected because the log records are sequentially written and does not yield the overwrite operation. Especially, the gap is about 9.23% when key values are fully sequential order.



(a) BFTL and BFLT with ERS

(b) IBSF and IBSF with ERS

(c) WOBF and WOBF with ERS

Figure 3. The Consumed time when inserting 100000 records

4.2. Performance of Recovery

Figure 4 presents the consumed time of recovery data when the systems reboot after crashes which randomly occurred. It can be seen that ERS quickly recover in all cases of rs. On average, it takes about 0.41 to 0.44 seconds to recover all the data which have not been written before the crashes occur. ERS consumes about 0.44 seconds for BFTL, 0.41 seconds for IBSF, and 0.42 seconds for WOBF to recover the data loss.

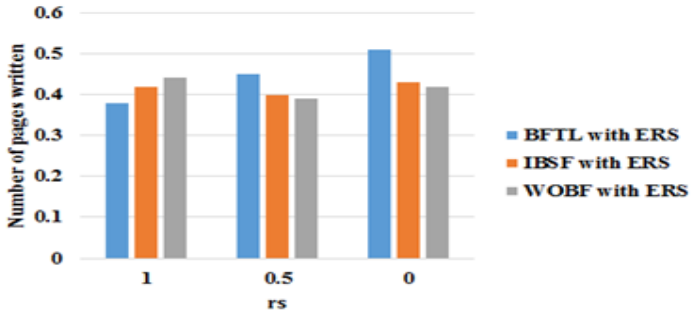


Figure 4. The recovery time

In fact, the consumed time for restoring the data does not depend on rs. Instead, it depends on the size of index buffer because ERS sets a commit record in the logger whenever the index buffer is successfully committed. This means if the size of index buffer is big, the number of records which needs to be recovered after a crash is large resulting in lots of time consumed for restoring the data.

Through these experiments, we can see that the performance of ERS is quite good. Besides that, it is much more reliable because it helps the buffer-based B-tree quickly recovers the data after a crash. Therefore, it will be good in practical systems.

5. CONCLUSION

Flash memory and B-tree index structure are widely used for embedded systems, personal computers, and large-scale server systems. Due to hardware restrictions, the performance of flash memory could significantly deteriorate when directly implementing B-tree. To solve this issue, many buffer-based B-tree index variants have been proposed for flash memory in order to reduce the number of flash operations. However, these B-tree indexes suffer from the risk of losing data when a sudden power-off occurs. In this study, we proposed a new recovery scheme for Buffer-based B-tree indexes on flash memory. The proposed system can minimize the loss of data by deploying logging and recovery policies. The experimental results show that ERS yields a good performance and helps the buffer-based B-tree indexes improve the reliability.

ACKNOWLEDGEMENTS

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2015R1D1A1A01056593)

REFERENCES

[1] Shinde Pratibha et al. “Efficient Flash Translation layer for Flash Memory,” International Journal of Scientific and Research Publications, Volume 3, Issue 4, April 2013

[2] E.gal, S. Toledo, “Algorithms and data structures for flash memory,” ACM Computing surveys 37, 2005, pp138-163

[3] D. S. Batory, “B+-Trees and Indexed Sequential Files: A Performance Comparison,” Proceeding of Special Interest Group on Management of Data, 1981, pp. 30-39.

[4] Chin-Hsien Wu et al. “An Efficient B-Tree Layer Implementation for Flash Memory Storage Systems,” ACM Transactions on Embedded Computing Systems, Vol. 6, No. 3, Article 19, 2007

[5] Hyun-Seob Lee and Dong-Ho Lee, “An Efficient Index Buffer Management Scheme for Implementing a B-Tree on NAND Flash Memory,” Data & Knowledge Engineering, vol. 69, no.9, 2010, pp. 901-916.

[6] Xiaona Gong et al. “A Write-Optimized B-Tree Layer for NAND Flash,” Proceeding of the 7th International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM), pp.1-4, 2011

[7] J. S. M. Verhofstad, "Recovery Techniques for Database Systems," ACM Computing Surveys, vol. 10, pp. 167-195, 1978.

[8] Drew et al., "A Comparison of File System Workloads," Proceedings of the 6th USENIX Conference on File and Storage Technologies, 2000, pp. 41-54.

[9] Andrew W Leung et al., "Measurement and Analysis of Large-Scale Network File System Workloads," Proceedings of the 6th USENIX Conference on File and Storage Technologies, 2008, pp. 213-226

AUTHORS

VanPhi Ho received his BS and MS degrees in the Computer Science Department at Da Nang University in October 2004 and October 2009, respectively. He is currently a Ph.D. student in the School of Computing at Soongsil University. His research interests include flash memory-based DBMSs and database systems.

Dong-Joo Park received his BS and MS degrees in the Computer Engineering Department at Seoul National University in February 1995 and February 1997, respectively, a Ph.D. in School of CS&E from Seoul National University in August 2001. He is currently an associate professor in the School of Computing at Soongsil University. His research interests include flash memory-based DBMSs, multimedia databases, and database systems.

Seung-Joo Jeong is currently a Master's course student in the School of Computing at Soongsil University. His research interest include flash memory-based DBMSs and database systems.