A Novel B-tree Scheme for Flash Memory

VanPhi Ho, Dong-Joo Park
School of Computer Science and Engineering,
Soongsil University
Seoul, Korea
{hvphi, djpark}@ssu.ac.kr

Abstract

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.

This study introduces a novel B-tree scheme and an index buffer management scheme, called BMS which works over FTL. BMS could reduce the number of read and write operations and minimize the number of pages used to store the B-tree by applying overflow mechanism and eliminating redundant index units in the index buffer. The experimental results show that BMS yields a better performance than that of the other existing variants of B-tree index.

Keywords: *B-tree index, flash-aware index, flash memory.*

I. Introduction

Flash memory [1-2] has been widely used because it has many positive consequences such as high speed access, low power consumption, small size and high reliability. Beside these advantages, it has some downsides including erase-before write, limited life cycle. To help hosts access data stored inside flash memory as accessing disk drives, a module called Flash Translation Layer (FTL) is used to process the logical-to-physical address mapping between the host and flash memory. The FTL [1] has two main features: address mapping and garbage collection. To date, some FTL algorithms have been proposed to confine physical limitation characteristics and improve the overall performance of flash memory.

Despite the fact that performance of a flash memory has been enhanced by using FTL algorithms, it may encounter performance degradation problems when implementing B-tree index directly because the overwrite operations on flash memory frequently occur in case of updating B-tree nodes.

This paper proposes a novel B-tree scheme and a new buffer management scheme, called BMS which uses a main-memory resident index buffer to temporarily store newly created index units. When the index buffer runs out of space, the first index unit and its related index units are selected, then the BMS builds a logical node and finally writes the node into the flash memory. BMS could not only reduce the number of write operations, minimize the number of pages used by applying overflow mechanism, but also reduce the number of commit operations by buffering index units and eliminating redundant index units in the index buffer.

The experimental results indicate that our proposed B-tree index structure achieves a better performance in comparison with the other existing B-trees.

The rest of this paper is organized as follows: Section 2 reviews basic knowledge of flash memory, flash-aware index structures and discusses the drawbacks of related works. Next, the design of BMS and its operations are presented in Section 3. Section 4 experimentally evaluates the efficiency of BMS, and finally, Section 5 concludes the paper.

II. Background and related works

Flash memory is a type of nonvolatile storage device that is widely used nowadays. Unlike a hard disk, flash memory consists of a number of NAND flash memory arrays, a controller and a SRAM. A flash memory is organized in many blocks and each block contains a fixed number of pages. A page is a smallest unit of read and write operations, while erase operations are handled by block. Flash memory requires an intermediate software layer called Flash Translation Layer (FTL) [1] for managing and controlling data by which the overall performance of flash memory can be enhanced.

B-tree index [3] is a structure that is widely used in many file systems and database management systems in consequence of quickly access capability. However, the frequent random writes of B-tree may degrade the efficiency of B-tree index on flash memory 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 composed of a reservation buffer and a node translation table. 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 many index units for the same node may be written in various pages, a node translation table is used to collect 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, it may generate many read operations to access a B-tree node because the data of one node may be scattered in different pages. Moreover, the node translation table and its list must be

maintained in RAM and its size may rapidly grow. Therefore, BFTL consumes a large amount of main memory. In addition, BFTL has a variety of redundant index units in the index buffer leading to that many write operations are performed.

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 into 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 a 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 redundant index units in the index buffer of IBSF, many commit operations and write operations are executed.

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 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 has a lot of redundant index units in the index buffer.

III. The design and implementation of BMS

A. The design of BMS

This section presents a new B-tree index structure and a new buffer management scheme, called BMS, to efficiently implement a B-tree on flash memory. Its objective is to significantly reduce the number of flash operations when building a B-tree and minimize the number of flash pages used to store the nodes of B-tree. BMS is a software module which can be used by any access module for the application using the B-tree on flash memory. It processes the B-tree related requests from upper layer and sends requests to FTL.

In order to achieve the aforementioned goal, we maintain a

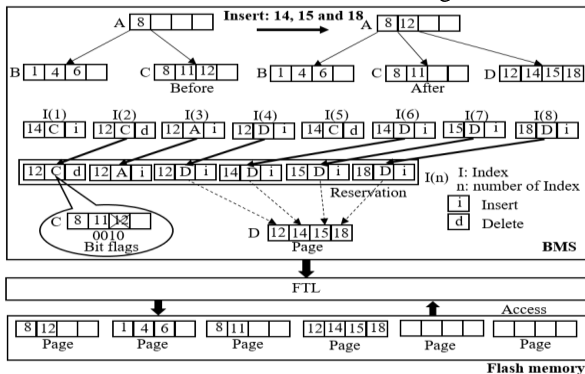

Figure 1. Overall architecture of BMS

main-memory resident index buffer which temporarily stores newly created index units and a B-tree module which is resided in flash to avoid losing data.

Figure 1 shows the architecture of BMS comprising an index buffer and a B-tree module.

The B-tree module adopts the overflow mechanism which does not split the leaf node if the leaf node is fulfilled by sequential key values. In BMS, instead of splitting the full leaf node whose keys are inserted sequentially, the leaf node is written into flash memory before the overflow occurs. This mechanism helps BMS reduce the large number of read, write operations. The index buffer is managed by the commit policy. It is used to store index units temporarily which reflect the modified nodes.

The index unit consists of components of the original B-tree node: primary_key, data_ptr, parent_ptr, right_ptr, and left_ptr. Besides the original components, the index unit has two more components that are node_identifier and type. The node_identifier denotes to which B-tree node the index unit is belonging. The type distinguishes whether the data is for the insertion or deletion.

The index buffer manages two type of index units: insertion_type and deletion_type. When the index buffer is full, commit operations are executed to flush some index units from index buffer to flash memory. Since BMS stores all index units for one node in one page and many redundant index units in the index buffer are eliminated, a large number of read and write operations are reduced when constructing B-tree.

B. The implementation of BMS

1. Insertion operation

When a record is inserted to B-tree, one or more index units are created to reflect the insertion. The processing is performed as follows: BMS checks if there is a redundant index unit which has the same primary key in the index buffer or not. In case of existing, BMS deletes the redundant index unit, and then inserts the newly created index unit into the index buffer. On the contrary, BMS checks whether the index buffer is full or not. If the index buffer is not full, BMS inserts the newly created index unit into it. By contrast, BMS performs a commit operation to vacate the index buffer, then, inserts the newly created index unit into the buffer.

A special case of the insertion operations is that all keys of a leaf node are inserted contiguously. This could be exemplified by inserting sequential records as shown in Figure 2. In this example, node D has 4 keys in which each key value in turn is 13, 14, 15, and 16. If a record with key = 20 is inserted into the general B-tree, the leaf node D is overflow, and in consequence of that, the leaf node D is split into two nodes. A split operation of a leaf node results in many read,

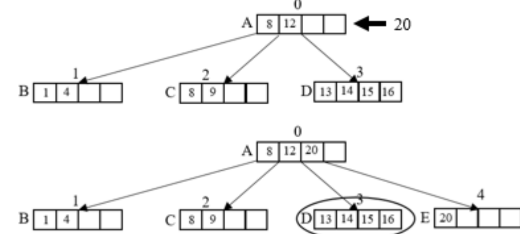

Figure 2. An example of sequential insertion

write and erase operations and it may lead to a serial split operation of the parent nodes in the general B-trees. Therefore, the number of flash operations increases significantly. BMS performs these insert operations in a different way applying overflow mechanism. BMS will store the whole node D into flash memory before the overflow occurs instead of splitting

the leaf node D. Afterwards, the new record is inserted into a new node E.

Processing insert operations under this order allows BMS to reduce a number of flash operations. Additionally, the page utilization increases because the utilization of the leaf is full in BMS. As about 80-90% of the write pattern is sequential in the practical file systems [7, 8], the performance of BMS in the practical systems will be enhanced.

2. Deletion operation

Similar to an insertion operation, when a record is deleted from B-tree, an index unit is created to reflect the deletion. However, the processing of deletion operation is a bit different from that of insertion operation due to the discrepancy in the structure between deletion_type and insertion_type index unit. For the deletion_type, it only needs the location of the entry in the node to reflect the deletion of entry in the B-tree node. Since the location of entries can be expressed as a bit flag set, the deletion_type index unit maintains a bit flag set to mark the entries to be deleted. Therefore, many deletion_type index units for one node can be stored in one index unit. This helps BMS reduce the number of redundant index units in the index buffer.

The processing of deletion operation is executed as follows: A deletion_type index unit is created when a record is deleted. BMS checks whether or not the index buffer has the insertion_type index unit which has the same key value as that of the newly created index unit. If so, BMS deletes the insertion_type index unit without inserting the newly created index unit (deletion_type index unit) into the index buffer. On the other hand, if there is a deletion_type index unit having the same node_identifier as that of the newly created index unit, BMS updates it by converting the bit flag set. Nevertheless, if there is not, BMS inserts the newly created index unit into the index buffer if the index buffer is not full. Otherwise, a commit operation is executed to vacate the index buffer and then the newly created index unit is inserted into the index buffer.

As shown in the figure 3, the index unit <14, C, i> is the redundant index unit when the index unit <14, C, d> is created.

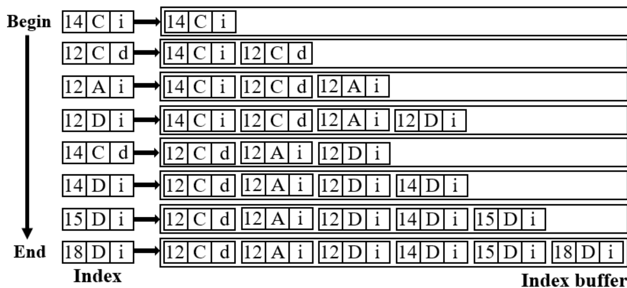

Figure 3. An example of the index buffer management in BMS

3. Commit operation

The newly created index units are temporarily stored in the index buffer. Owing to the limitation of the index buffer size, BMS has to flush index units from the index buffer to flash memory when the index buffer is full by using commit operation. The commit operation is as follows: First, BMS chooses the first index unit in the index buffer and collects its related index units. Next it reads the page associated with this index unit, builds the logical B-tree node and finally writes the node into flash memory.

Algorithm 1 presents detail of the commit operation. In line 1, BMS gets the first index unit in the index buffer. In line 2 to 8, BMS collects all index units which belong to the same node with the first index unit. After collecting index units, BMS reads the logical node related to the victim index units, reflects victim index units into the logical node and then writes the logical node into flash memory as displayed from line 9 to 11.

Algorithm 1. Commit operation

1: Input: New_index_unit
2: Output: none
3: *v_identifier*=the first index unit identifier in the buffer
4: **for** *i=2 to sizeof(index_buffer)*
5: **if** *v_identifier* is the same as the identifier of *index_unit(i)*
6: collect *index_unit(i)* as victim index units
7: **end if**
8: **end for**
9: read the logical node related to the victim index units
10: reflect victim index units into the logical node
11: write the logical node into flash memory

Since redundant index units in the index buffer are eliminated, BMS reduces the number of index units in the index buffer. Consequently, BMS reduces a large number of commit operations and write operations on flash memory.

IV. Performance evaluation

This section shows the experimental results achieved by applying the proposed BMS and compares its performance to that of the original B-tree, BFTL and IBSF. 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 B-tree, BFTL, IBSF and BMS were assessed in terms of performance metrics: the number of pages read, the number of pages written and the number of blocks erased, the average time to build trees and the number of block used. In order to control the key value distribution, a ratio called rs (ratio of 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.

A. Performance of the B-trees creation

In this section, we assess the performance of read and write operations and time consumption when building B-trees. Figure 4 and Figure 5 show the number of pages read and written when inserting records by varying the value of rs. Overall, BMS yielded a better performance than those of IBSF, BFTL and the original B-tree in all cases. In BMS, since it did not split leaf nodes frequently in cases of the higher rs, the great number of read and write operations were reduced. Therefore, the performance of BMS was much better than those of the others. In Figure 4, the performance of BMS was average enhanced by 12% compared to that of IBSF, 18%

compared to that of the original B-tree and 76% compared to that of BFTL.

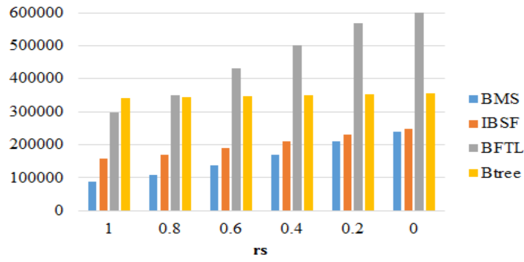

Figure 4. The number of pages read

For the write performance, as shown in Figure 5, BMS improved by 19% compared to that of IBSF, 22% compared to that of BFTL and 56% compared to that of the original B-tree. BMS achieved a better write performance than that of the others because it efficiently applied overflow mechanism, used the index buffer and eliminated the redundant index units.

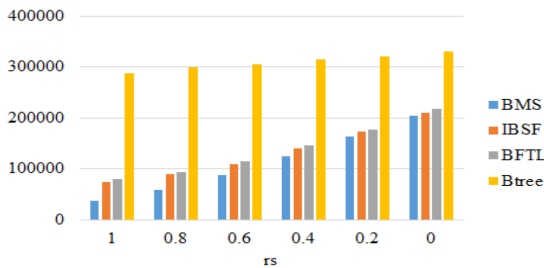

Figure 5. The number of pages write

Figure 6 presents the consumed time when constructing the trees. Since using overflow mechanism and eliminating redundant index units, BMS built the tree faster than the others in general. When the inserted keys were in ascending order, the performance of BMS was improved by 80% compared to that of BFTL, 92% compared to that of IBSF and 196% compared to that of the original B-tree. In contrast, when the inserted keys were in random order, the performance of BMS was improved by 4% compared to that of BFTL, 9% compared to that of IBSF and 46% compared to that of the original B-tree.

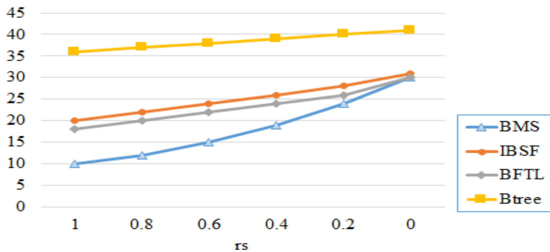

Figure 6. The average consumed time

B. Page utilization

This section depicts the results of experiment based on page utilization to assess the overflow mechanism. Figure 7 illustrates the number of flash blocks needed to store B-trees index when 100,000 records were inserted. It is apparent that the number of blocks that BMS used to store index was less than that of other trees in all cases. As the ratio goes to 1, BMS needed just by half number of blocks used compared to the other trees did because it did not perform the node split in the case of sequential insertions. In contrast, the leaf nodes of the other trees are always half full because of splitting nodes, so they needed more blocks than BMS.

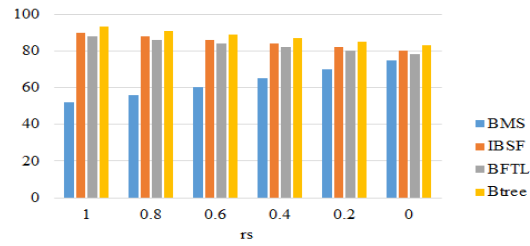

Figure 7. The number of blocks used

V. 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. In this study, we proposed a new buffer management scheme. The proposed system not only helps to improve the performance of flash memory but also enhances the page utilization of the flash memory by using the overflow mechanism. The experimental results showed that BMS yields a better performance than that of the other trees.

Acknowledgment

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

[8] 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