PAPER

CAWBT: NVM-Based B+Tree Index Structure Using Cache Line Sized Atomic Write

Dokeun LEE[†a)], ***Member***, **Seongjin LEE**[††b)], ***and*** **Youjip WON**[†††c)], ***Nonmembers***

SUMMARY Indexing is one of the fields where the non-volatile memory (NVM) has the advantages of byte-addressable characteristics and fast read/write speed. The existing index structures for NVM have been developed based on the fact that the size of cache line and the atomicity guarantee unit of NVM are different and they tried to overcome the weakness of consistency from the difference. To overcome the weakness, an expensive flush operation is required which results in a lower performance than a basic B+tree index. Recent studies have shown that the I/O units of the NVM can be matched with the atomicity guarantee units under limited circumstances. In this paper, we propose a Cache line sized Atomic Write B+tree (CAWBT), which is a minimal B+tree structure that shows higher performance than a basic b+ tree and designed for NVM. CAWBT has almost same performance compared to basic B+tree without consistency guarantee and shows remarkable performance improvement compared to other B+tree indexes for NVM.

key words: *non-volatile memory, key-value store, index structure, B+tree*

1. Introduction

The introduction of NVM [1]–[4], that provide persistency and have comparable access speed of DRAM, have initiated novel approaches in NVM file systems [5]–[8] and persistent heaps [9]–[12]. There are substantial gains in exploiting NVM in both of the research fields because file system metadata, database logs, and indexes which are small in size are not only accessed frequently but also have to maintain its persistency.

Since index plays significant role in improving performance in file systems, key-value stores, and databases, the research community have proposed various B+tree based index structures for NVM [13]. However, there is a critical issue in ensuring consistency while storing index structure in NVM. Since CPU cache is volatile the issue is inevitable. In order to make the index structure consistent with the original data in the system, cache lines in the CPU must be flushed with `clflush` or `mfence` commands. Typical latency of `clflush` is known to be 25ns and it is reported that theses commands greatly degrades the I/O performance of NVM [14].

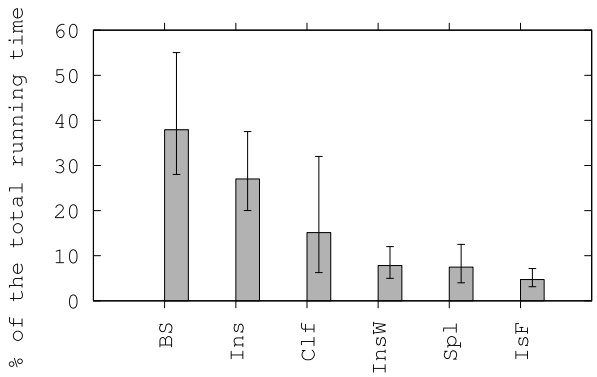

Fig. 1 The percentage of total insertion time in basic B+tree (BS: Binary_search(), Ins: Insert(), Clf: `clflush`(), Insw: Insert_Wrapper(), Spl: Split(), Isf: Is_Full())

Figure 1 shows the average percent of the time taken by each function while inserting 500k key value pairs to a B+tree with `clflush`. The performance is measured with gprof [15] on a DRAM based system. The result shows that `binary_search()` and `clflush()` spends about 38% and 15% of the total time, respectively. The result clearly shows that it is important to optimize read operations in NVM and reduce the number of `clflush`.

To provide consistency for B+tree based index while maximizing the performance of NVM, we need to exploit atomic write and decide the size of a node such that it does not call additional operations. The use of atomic write is a quite common practice in providing consistency. Along with WB+tree [16] many works are based on the fact that the size of atomic write is 8 bytes [6], [7], [17]–[20]; however, evidently 8 byte is too small in size to store all the important information of a node. To address the issue of the size of atomic write, some works divide a node into two where only the essential part of the node is stored in NVM for the consistency. In the case of sorting in WB+tree, it exploits 8 byte atomic write to decrease the number of writes to NVM; however, it has adversary effect of increasing the number of reads to fetch the rest of data of a node. To decide the size of a node, we need to make sure the issued size of read and write operation is aligned to cache line, the size is not too small to invoke split operation, and also not too large to increase the time of binary search operation.

In this paper, we introduce CAWBT to address the issues. In order to maximize the input/output performance of

Manuscript received February 4, 2019.
Manuscript revised July 17, 2019.
Manuscript publicized September 12, 2019.

[†]The author is with the Department of Computer Software, Hanyang University, Seoul, Korea.
[††]The author is with the Department of Aerospace and Software Engineering, Gyeongsang National University, Jinju, Korea.
[†††]The author is with the School of Electrical Engineering, KAIST, Daejeon, Korea.
a) E-mail: matureelf@hanyang.ac.kr
b) E-mail: insight@gnu.ac.kr (Corresponding author)
c) E-mail: ywon@kaist.ac.kr (Corresponding author)

DOI: 10.1587/transinf.2019EDP7034

B+tree, we optimized insert and search operation by tailoring the size of the operation to the size of cache line. Essential portion of a node in index to maintain consistency is stored in NVM and the rest of the node is kept in DRAM. We employ cache line sized atomic write to make a portion of a node in NVM is consistent as well as to increase the size of data of a node in NVM. We further employ minimal logging and recovery mechanism to ensure all the data is consistent even in a split operation where it requires more than a cache line sized atomic writes.

2. Background and Related Work

This section describes basic techniques for consistency guarantee in NVM and their problems and discusses approaches to existing techniques for solving problems.

2.1 Consistency Guarantee in NVM

The `clflush` and `mfence` commands are required to ensure consistency between the volatile CPU cache and the NVM. `clflush` is used to force the value stored in the dirty cache line into memory, and `mfence` is an instruction that prevents the order of memory access instructions from being changed due to CPU reordering. `clflush` and `mfence` must be used at the same time to ensure complete consistency, and these two commands are generally known as expensive. B+tree based index structure invokes many additional NVM writes during the split operation and during sorting intra nodes. A single atomic write is not sufficient to keep consistency because the size of the NVM write can be larger than the failure-atomicity unit.

Traditional methods for solving this problem are logging and copy-on-write (CoW). Logging ensures consistency by recording changes in the node and redo or undo the transaction, and CoW guarantees consistency by creating a copy and updating only the pointer that accesses the copy. However, these techniques also require additional NVM writes, which degrade the overall I/O performance of the index structure. The slow NVM write performance is coupled with `clflush` which is used for maintaining the consistency, resulting in more substantial performance degradation.

2.2 Data Structures for NVM

2.2.1 CDDS B-Tree

The CDDS b-tree [14] is a b-tree that records a node's updated version when keys are inserted to the nodes. When the key update, the node is not overwritten. Instead, CDDS b-tree makes a new copy of the node with a new version number. The old node becomes garbage, and it is maintained for consistency and recovery. When a crash occurs during the update, it is possible to recover using the previous version of the data. Providing consistency through version control is the most significant advantage of this data structure; however, there is garbage collection overhead and it also does not offer `clflush`-related optimizations. For those reasons, the performance of CDDS b-tree is lower than other data structures for NVM in terms of insert and search operations.

2.2.2 NV-Tree

NV-Tree [21] maintains all non-leaf nodes in DRAM space. These nodes are created consecutively in memory space, and NV-Tree does not consider the consistency for non-leaf nodes for performance. Instead of that, NV-Tree maintains leaf nodes in NVM for the consistency and uses the append-only update for performance of insert operation. NV-Tree uses `clflush` only for leaf nodes, and there is no sorting of leaf nodes; consequently, the insertion performance is improved. However, because NV-Tree maintains pre-allocated index, it is possible to overflow when the number of inserted key exceeds the allocated size. When overflow occurs, all non-leaf nodes are deleted, and a new index is reconstructed; thus, whole index performance is degraded.

2.2.3 wB+Tree

wB+tree [16] uses append-only update to prevent the sorting of intra node key, which is the operation that generates flushes the most in B+tree structure and proposed new metadata called slot array to store the order key. The consistency is guaranteed by using the 8-byte atomic update and indicating updated slot by exploiting the bitmap. wB+tree uses append-only updates, but because the slot array knows the order of keys for nodes, it is possible to handle a range query. wB+tree reduced the number of flushes compared to previous NVM data structures; however, it still requires at least four flushes to update a node. wB-Tree did not propose a particular technique for split operations, and it has overheads to access the slot array and bitmap.

2.2.4 FPTree

FP-Tree [19] is data structure that maintains non-leaf nodes in DRAM while leaf nodes are kept in NVM like the NV-Tree. FP-Tree maintains the concurrency while accessing a non-leaf node using hardware transactional memory by allocating non-leaf nodes to DRAM. This data structure suggests a way to reduce cache miss ratio through fingerprinting. Although FP-Tree performs better than NV-Tree, there is still overhead of reconstruction like NV-Tree in case of a system crash.

2.2.5 HiKV

HiKV [13] is a KV-store consisting of a hybrid structure of hash and B+tree. The KV operations like put, get, and update has excellent performance in hash. However, the scan (range query) operation cannot be performed in hash; therefore, hash cannot be used for various DB and KV indexing structures. The B+tree has a good structure for scan,

but it has poor performance in NVM because of key sorting. HiKV places the B+tree in the DRAM and arranges hash in the NVM so that operations that have a favorable hash such as put and get are processed directly by the NVM, and the scan is processed by b+ tree. The goal of HiKV is to increase the performance of processing single instruction to handle many concurrent transactions of hybrid indexes, and to minimize the overhead of maintaining consistency between two structures. To solve this problem, HiKV designed dual structures of serving thread and backend thread. The serving thread handles the put or get operation in hash and handles scan operation in B+tree. The backend threads move data inserted in hash into B+tree. HiKV has proposed a dynamic thread adaption scheme that dynamically adjusts the number of threads for many concurrency processes, and introduces key-based hash partitioning. To ensure the consistency, B+tree in HiKV exploits using hardware transactional memory (HTM), and lock per partition for hashing.

2.2.6 Fast-Fair Tree

Fast-fair tree [22] is a B+tree that acts as a mechanism to allow node inconsistency temporarily. They mandate that duplicate keys are not allowed, and if duplicate keys are detected during insert or search operation, they are regarded as inconsistent state. The inconsistent node can be recovered back to the consistent state through shift operation without a forced flush, but detecting duplicate keys for every search operation can adversely affect index performance.

2.2.7 Multi-Word Atomic Update

Wang et al. designed a lock-free index using the PMwCAS [23], which can atomically update multi-words. PMwCAS is similar to a software transaction. This technique manages the collision of multiple threads by tracking the status of the memory addresses (operation, old value, new value, dirty bit, and so on) and then atomically applying compare-and-swap. PMwCAS provides APIs to developers to manage the NVM layer easily; however, this technology still has problems with software transactions. The memory read/write overhead for managing the descriptors is obvious and this weakness is more noticeable in NVMs, which have relatively slow performance compared to DRAM.

2.3 64 Byte (Cache Line) Atomicity

Most of the existing NVM index structures have been developed based on the fact that memory I/O unit (cache line, 64 bytes) and failure-atomicity guarantee unit (8 bytes) are different. However, in the part of recent studies, it is claimed that the failure atomicity unit may be larger than the 8 bytes. In particular, PMFS [7] mentioned that cache line sized atomic writes are possible by using Restricted Transactional Memory (RTM). In Strata [24], the metadata consistency guarantee unit is designed to be 64 bytes, which implicitly indicate that the consistency guarantee unit is 64 bytes.

3. Design

Operations such as `clflush` and `mfence` are expensive but inevitably used for consistency of NVM. Therefore, to improve the performance of B+tree index in NVM, it is essential to reduce the frequency of flush regarding writing. It is also important to optimize the read because the reads are the most significant part of the overall operation. In this study, the primary purpose is to achieve both of the optimizations at the same time and to use the hardware performance of NVM as much as possible. In this paper, we focus on the B+tree structure optimization using cache line sized atomic write. If the failure-atomicity unit is a unit of a node of B+tree, the number of flushes can be reduced to a minimum because there is no need for multiple flushes in sorting the keys in a node. In this case, search performance (read) is almost close to the basic B+tree's search performance, since `mfence` is not needed and no additional metadata is needed to ensure consistency. As discussed in Sect. 2.3, recent studies have shown that cache line sized atomic writes are feasible.

In this study, we optimized the size of one node of B+tree to 64 bytes based on recent studies. 64 byte (size of the cache line) is a small size for representing the necessary information of one node of B+tree, but the minimum information can be expressed. 64 byte is a size that can barely represent information of B+tree of degree 3. When the degree is 3, the split operation occurs at least once every two key inserted in the B+tree. Since the split operation is the most expensive in the B+tree where write occurs on at least three nodes, performance degradation due to the increase in split operation's count may be larger than gain obtained by minimizing the consistency guarantee cost for the low degree. In this study, CAWBT reduces the size of information which is written to one node to prevent this problem. For this purpose, we designed specific techniques such as flag embedding and the ID table.

Another problem with split operation is that the amount of information that needs to be written to NVM is larger than the cache line size of atomic writes. This means that the consistency guarantees for split operation cannot be done with one atomic write; therefore, another consistency guarantee technique is needed. Previous research have not provided a specific solution for ensuring consistency during split operation. In this study, we focused on the fact that the information required for redo recovery can be contained in one 64byte atomic write. We have developed a minimal logging technique to minimize the overhead in logging base on that fact. The design goals in this paper are summarized as follows. First, minimization the use of `clflush` and `mfence` by exploiting cache line sized atomic write. Second, node structure optimization for minimizing the occurrence of logging due to the increase of split operation. Third, the optimization of logging structure to obtain fast performance. Fourth, the recovery method for maintaining the consistency of the entire data structure. Fifth, the optimization of an in-

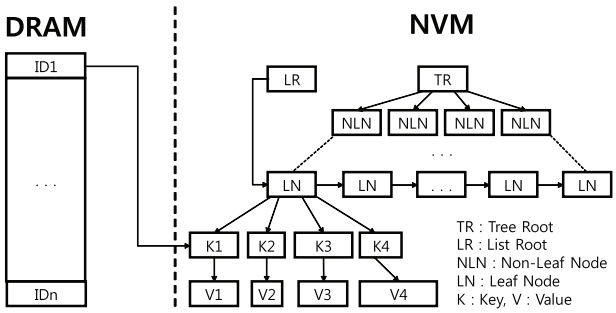

Fig. 2 Index structure

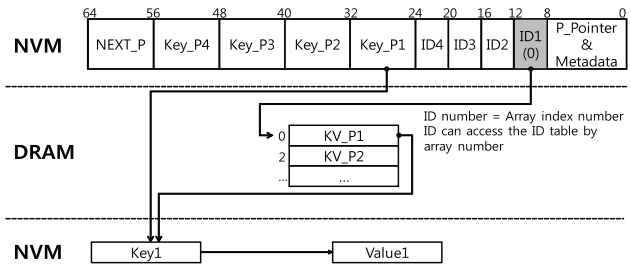

Fig. 3 The relation between the leaf node and the ID table

tra node key sorting for fast insertion/search. Sixth, the optimization of the B+tree search algorithm for reducing memory access.

3.1 Consistency Model

The consistency of NVM index structure is as important as the performance, so it must meet the ACID property like in a database. However, this study does not support durability because we are dealing with index structure rather than standalone KV store. The operation which changes data in B+tree index mostly is update at a node and split operation involving at least three nodes. The cache line size atomic write guarantees both atomicity and consistency in the intra node update. Split operations cannot guarantee atomicity and consistency with a single atomic write because multiple nodes are updated during the operation. In this case, we guarantee the atomicity in each updates of node using cache line sized atomic write, and the whole split operation maintains consistency using redo logging.

3.2 Index Structure

3.2.1 Overall Structure

The overall structure of the index is shown in Fig. 2. There is a B+tree structure in NVM, which is main index structure and in DRAM, there is a mapping table linking ID and KV pair. ID is registered in ID table and is used in nodes instead of a large-sized key. The ID is required to reduce the number of the split operation. The key can be found through the ID table, and all operations which are referring to the key will access the ID table. If the ID table is placed in the NVM, it affects I/O performance of index because it incurs the consistency guarantee cost (`clflush` and `mfence`). Therefore, this table is placed in DRAM and is designed to be reconstructed when the system crash occurred. Through this, CAWBT can eliminate the consistency guarantee cost while reducing the number of split operations.

Figure 3 shows the linking between the leaf nodes and the ID table. All non-leaf nodes always references the ID table to access the key. However, unlike a non-leaf node, a leaf node stores a pointer to an actual key in the space which stores a child node pointer. Since the table is in DRAM, we use this design to recover the ID table when data is lost. For the recovery of ID table, the leaf node is designed as a dual structure that can access the KV pairs through the ID and the pointer stored in the node.

3.2.2 Node Structure and the ID

The main feature of CAWBT is use of cache line size atomic write for intra node updates. If the size of a node of the B+tree matches the size of the cache line, consistency can be guaranteed with single `clflush` without additional `clflush` that is inevitably used to ensure consistency in the key sorting of intra node. This case is theoretically the best update speed for a node except for the case where there is no `clflush`, because it guarantees consistency at the same time as the update of the node.

The challenge here is that split operation frequently occurs due to small size of nodes. Split operations require additional consistency guarantees costs because they exceed the size of writes that can be guaranteed with atomic writes of a single cache line. They cause additional NVM writes and `clflush`/`mfence`, so frequent split operations degrade the overall performance of index.

The number of split operations is proportional to the degree of tree. The maximum degree of tree is 3 (parent pointer 1, key 3, child pointer 4) because 8 bytes are needed to represent one address based on the 64-bit computing environment, and generally, 8 to 16 bytes size is used. The five pieces of information representing the address make it difficult to reduce the size, and there is no significant benefit even if the address bit is reduced because the redirection is expensive.

In this study, the number of bits that represent key is reduced to increase the degree of tree by one. For this purpose, the number of nodes is increased by one by using the 4 byte ID instead of the 8byte Key. Because one child pointer and one ID are needed to increase the degree by one, the degree that can be increased by 4-byte ID is maximum 4. Since the number of bits expressing the address is always fixed, it is difficult to increase the degree. Since the size of the ID is closely related to the range of the expressible key, if the ID is small, there is a problem with the expandability of the index structure. 2 byte ID can represent 65536 keys, and 3-byte ID can represent about 16.78 million keys, which is too low to cover the scope of large databases used in the industry. Increasing the degree to 4 can improve the index's

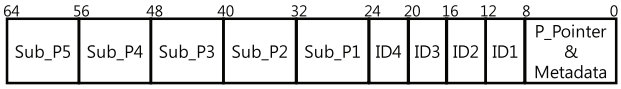

Fig. 4 The structure of non-leaf node

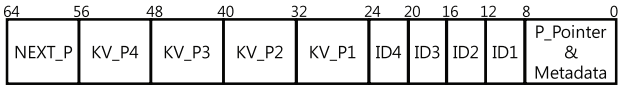

Fig. 5 The structure of leaf node

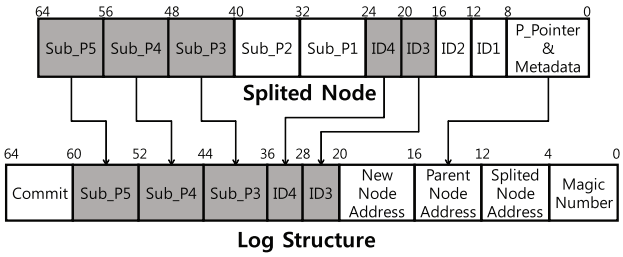

Fig. 6 Log structure

insert/search capabilities, and it can provide enough range. The experiment shows that the number of splits decreased by about 167% when the number of nodes was increased from 3 to 4 in 0.5M nodes insertion.

The structure of the non-leaf node and leaf node designed based on this is shown in Fig. 4 and Fig. 5. One non-leaf node is 64 bytes, and it includes six pointers and four IDs. In a 64-bit computing environment, a pointer occupies 8 bytes, and an ID occupies 4 bytes. P_Pointer& Metadata is the area where the pointer to the parent node coexists with the leaf bit, and the ID is recorded in the ID table and acts as an intermediary of the key. Sub_P is a pointer to a child node. The basic structure of a leaf node is the same as for a non-leaf node, and the location of the actual KV pairs is recorded instead of the child pointer.

3.2.3 ID Table

In the basic B+tree structure, the key is stored in the non-leaf node and serves as a separator in the search operation. As discussed above, CAWBT uses the IDs instead of keys to reduce the split count. In this structure, there is an advantage to support a variable key size naturally. Since the ID is used instead of the key as a delimiter of the node, a mapping table in which the ID corresponds to which key is required. In this study, we did not think that this mapping table must be in the NVM space. The reason is that placing additional tables in NVM is costly; it requires more expensive NVM space, and if the table is placed in the NVM, the I/O performance of the entire index is deteriorated because it is necessary to use a consistency maintaining technique. Since NVM has lower I/O performance than DRAM, this will also degrade the performance of the index. This table is a structure that can be re-built through leaf node traversal, so there is no reason to be in NVM.

The ID table is designed as a fixed array to improve access speed. The ID table is a data structure that is accessed frequently in all operations of the index; thus it is very important to make it as concise as possible. For a more efficient spatial management of the ID table, the number of memory access (‘*’, reference) per binary search is doubled (1 ID, 1 pointer) when designing with a variable data structure such as linked-list. It is because the memory reference operator ‘*’ always access memory even if the value is in cache already. In a fixed array, the ID is the same as the array index, so it can access the memory only once.

If a new insertion occurs, the ID must be assigned to the key to be stored. Even if there is an empty slot in the ID table, it is ignored and assigned to a slot that is larger than the recently assigned ID (slot number). In other words, the number of inserts and ID are the same, and the ID is assigned by the insert number counter. If the number of inserts exceeds the maximum number of the ID table, ID is assigned by searching for a list in which previous deleted IDs are recorded. The reason for this design is that the ID table can have a large size according to the range of the key, because the search of the ID table for assigning the ID also adversely affects the insert performance.

3.3 Fast Search Algorithm

Since CAWBT accesses the key in the ID table, it is possible to access the key without having to follow pointers to a leaf node. That is, the IDs existing in the non-leaf nodes can be found without searching through until leaf nodes. In an experiment to search 10 million KV pairs, 33,350,913 keys are quickly accessed through fast search algorithm and the other 6,649,098 keys are searched at the leaf node. The experiment showes about 12% improvement in performance compared to a general search.

3.4 Minimum Redo Logging

When split operation occurs, the amount of data to write exceeds 64 bytes, so it cannot maintain consistency using the cache line sized atomic write. Therefore, the split operation has to use CoW or general logging techniques. Among the logging techniques, undo logging requires at least two nodes of the source node and the parent node to be logged, so it writes at least 128 bytes of data to the log. In this case, the logging data cannot be written with the single `clflush`, which degrades logging performance. Also, it is hard to support atomicity in continuous split operations through undo logging. In the case of redo logging, the only data needed is 32 bytes of data (two IDs and three-pointers to child nodes) and 24 byte pointers (pointers to the source, the parent node, and the new node to be created). These data can be written to the log with the single `clflush`; therefore, redo logging can improve the performance of logging. Based on these facts, we designed a 64 byte log structure by adjusting the size of the log header, as shown in Fig. 6. As shown in the figure, the data copied from the split node are ID 3, 4, child pointer

3, 4, 5, and the addresses of the three nodes involved in the split operation. In the case of a continuous split operation, it is likely that multiple log records are needed, but since there is redo logging, only one log record exists in one split operation. This is an advantage of simplifying the logging algorithm. If a crash occurs during a sequential split operation, CAWBT only performs a restore operation on the node that crashed and continues with the next split.

At the start of logging, CAWBT records the magic number first and proceed with logging. When the index restarts due to a crash, CAWBT checks the status number in the log header. If it is 0, no crash occurs. If the status number is recorded in the log header, CAWBT considers the crash happens. In this case, the split operation will be restarted using the data recorded in the log.

3.5 Concurrency Control

CAWBT controls concurrency with two policy; the policy for read concurrency and the policy for write concurrency. The search is a read oriented operation, and insert and delete are write oriented operations. In the case of read operations, CAWBT does not have any special control over threads because it is not necessary to adjust the order of operations of threads even when multiple threads access the index.

In the case of write operation, however, it needs to operate in a more conservative point of view considering the persistency of NVM. When multiple threads try to write, only one thread modifies the index through global lock. In CAWBT, the fanout of the tree is small because one node is an extremely small index. If the fanout is small, the height of the tree grows, which means that a continuous split operation increases the number of nodes that need to be accessed and modified. In node-specific locking such as Masstree [25], CAWBT affects the latency of other threads because one thread can occupy the node with the maximum height of the tree, it is inefficient.

3.6 Key Size for Performance

Unlike other data structures, CAWBT structure supports variable key sizes and has stable performance regardless of the size distribution of key and value. Because the key is fetched through the ID table, there is no time difference in accessing the key other than the time to allocate space for key storage or to read and write key. In addition, in the user level memory allocator such as `malloc()`, since the memory allocation time according to the request size does not show a significant difference, the performance of the index is not different according to the size of the key and the value.

4. Implementation

4.1 Basic Operations

CAWBT implemented based on the bare-bone B+tree, except for the additional operations that required due to the change of the node structure. One of the differece is the bit operation that hides the flag bit which represents leaf. An other one is ID table references which occurs in all B+tree operations. Like the recent researches, the CAWBT maintains the consistency inside one node through forced flushing (`clflush`). In the case of split operations that can not be covered by a single atomic write, the CAWBT maintains the consistency through minimum logging.

4.2 Optimization of the CAWBT Node Structure

The basic operations of the B+tree require information like keys, addresses of child nodes, address of the parent node, flag which represents the leaf node, address of the sibling node, and the number of keys inside node. Since 64 byte node is too small for storing all the information, we removed the optional fields optimization of node structure. The address of the sibling node is used only in the traverse, and the number of keys in the node is obtained by counting the number of non-zero keys; therefore, we removed the two fields. The flag which represents the leaf node requires only 1 bit; however, at least 1 byte (char) must be allocated for memory allocation in the C language, thereby wasting space. This problem is solved by the flag embedding which hides the flag inside the address of the parent node exploiting bit coding. The 64-bit address can store the leaf flag bit in this space because the end of the address is always ending with 0x000. Because flag embedding requires decoding, there is a small overhead on index performance. However, since the parent's address is used only in the split operation, the performance degradation due to decoding does not have a significant effect on performance, and 1-bit coding is very fast.

4.3 Optimization for Searching in a Node

Binary search for intra-node alignment occupies a large part of the execution time of B+tree operations. In this paper, we have made several optimizations to reduce the execution time of an intra-node search. The first is to use the linear search instead of binary search. In general, linear search is faster than binary search in small degree of B+tree. The second is an optimization for fast search. In search algorithm of basic B+tree, there is no need to compare whether key is the same as one in the non-leaf node because real key exists in leaf node. However, in the fast search algorithm, the comparison is required even in non-intermediate nodes. To reduce the counts of the comparison, CAWBT does not compare when the key is lower than the one in the non-leaf node. When the key is not lower than the one in the node, CAWBT starts the comparison and post-process which is for fast search.

4.4 ID Table Re-Building

CAWBT guarantees atomicity and durability through the cache line sized atomic write and the minimal logging when

updating the index structure in NVM. However, since the ID table exists in the DRAM, it must be restored when the system power is shut down. Recovery is to newly input the key/value which stored in the leaf node into the table, and the specific area of the NVM space points to the head of the leaf node so that the leaf nodes can be scanned.

5. Experiments

5.1 Experimental Environment

The experiment was conducted on a system consisting of Intel (R) i7-3770 (3.40GHz), 6GB DRAM and Linux 2.6.32 environment. CAWBT is an in-memory index structure, thus all keys are stored in DRAM. CAWBT is compared with wB+tree [16], fast-fair tree [22], basic B+tree and basic B+tree with `clflush` and redo logging. We implemented the wB+tree as closest as possible based on their paper [16], and we use the source in the github.com in the case of the fast-fair tree. We do not use any optimization option in the compiler. For each index structure, we measured the performance of inserting and searching operations along with the performance of logging operation in performing split. The performance of recovery operation is also measured. The workload used in the experiments is the insertion of the 10 million KV pairs sequentially and the searching of all the 10 million KV pairs. In the other experiment, we used index with 0.5M KV pairs.

5.2 Verification

To verify that CAWBT is correctly implemented, we confirmed the number of split operation of CAWBT, B+tree, and wB+tree. Because of the characteristic of B+tree's split algorithm, the number of split operations must be the same if same workload is used in same degree. Figure 7 shows the number of splits in three data structures while inserting 0.5M KV pairs. Due to the characteristic of the B+tree, the data structures of the same degree must have the same number of the split operation. The result of the above graph shows that this technique is implemented correctly.

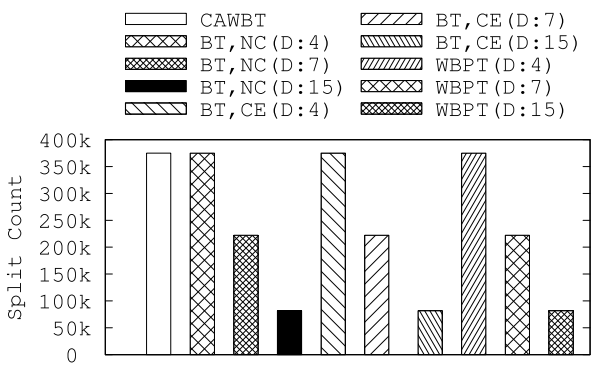

Fig. 7 Comparison of split count between data structures (CAWBT: CAWB+tree, BT: B+tree, NC: no consistency, CE: consistency ensured, WBPT: wB+tree, D: degree)

5.3 The Comparison of `clflush` Counts

Figure 8 shows the number of flushes for each data structure with different degrees in the comparison group. Considering along with the insertion performance (Fig. 9) which will be discussed in the next section, the `clflush` count tends to proportional to the index insertion performance. In addition to `clflush`, there are factors such as the split count according to the degree, and the overhead of the search inside of a node according to the degree.

5.4 The Comparison of Insertion Performance

Figure 9 shows the total time taken to insert 10 million sequential KV pairs. Since degree of fast-fair tree and CAWBT cannot be changed, we experimented with the fixed degree to make the comparison fair. CAWBT took 4,126ms to complete the task and shows the fastest performance among the entire comparison group. Since CAWBT have to refer to the ID table, it should show similar or less superior performance compared to basic B+tree. However, it performs faster than basic B+tree by optimization of intra node search. Overall, as the degree increases, the insertion performances of indexes become better due to the re-

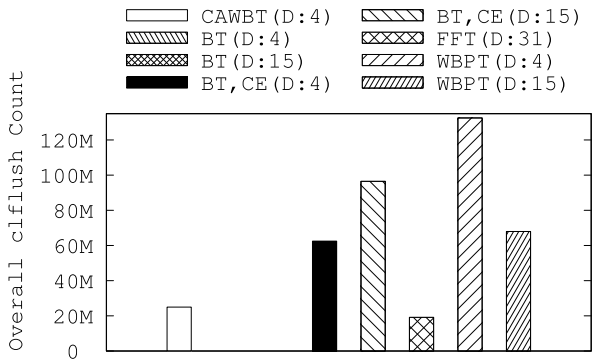

Fig. 8 Comparison of clflush count between data structures (CAWBT: CAWB+tree, BT: B+tree, CE: consistency ensured, WBPT: wB+tree, FFT: fast-fair tree, D: degree

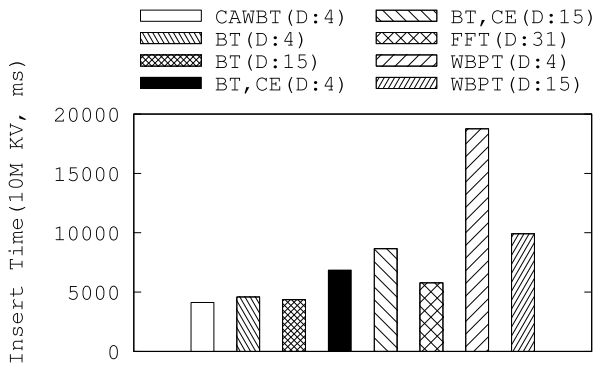

Fig. 9 Comparison of overall insertion time (10 M KV pair) between data structures (CAWBT: CAWB+tree, BT: B+tree, CE: consistency ensured, WBPT: wB+tree, FFT: fast-fair tree, D: degree)

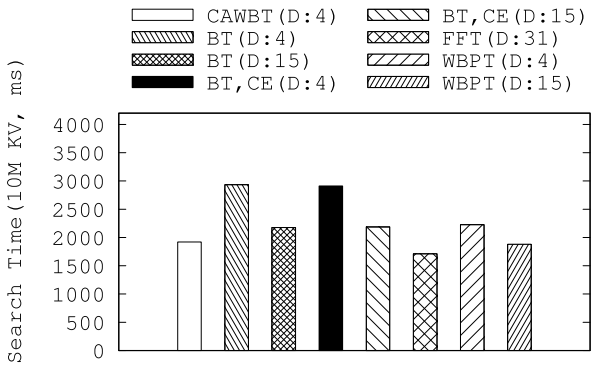

Fig. 10 Comparison of search time (10 M KV pair) between data structures (CAWBT: CAWB+tree, BT: B+tree, CE: consistency ensured, WBPT: wB+tree, D: degree)

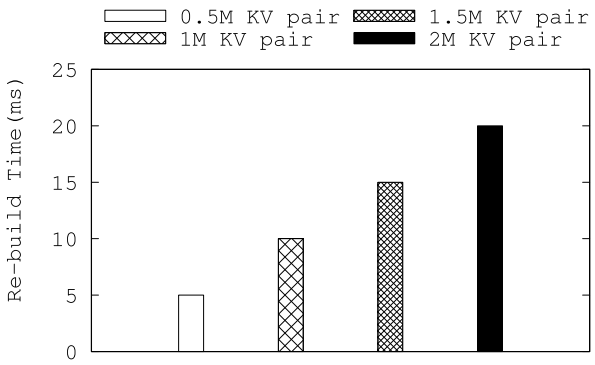

Fig. 11 Re-building overhead

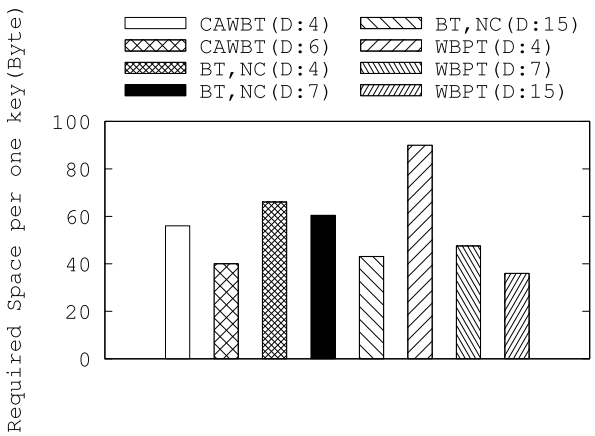

Fig. 12 Space overhead

duction of the split operation. However, in the case of the consistency-ensured B+tree, the insertion performance becomes worse because `clflush` is used whenever keys inside node are sorted, and because the amount of data which is written in log during split operation is increased. In the case of wB+tree, even though only one `clflush` is used for the key sorting within a node, there are two flushes to update the slot array and bitmap. Therefore, when the degree is small, number of `clflush` caused by logging is added to `clflush` needed for a node update so that `clflush` is generated the most (see Fig. 8). However, as the degree increases, it shows a drastic improvement in performance.

5.5 The Comparison of Search Performance

Figure 10 is a graph showing the sum of the all KV pair's search times in index which has 10 million KV pairs. Except for the CAWBT, the search operation in other data structures are performed until leaf node. Because the search operation is a sum of reads and compares, it is read-intensive operation. Also, there are few writes during the search operation. In this case, the most optimized structure for intra node search shows the highest performance. Fast-fair tree showed the best search performance, followed by wB+tree of degree 15. Overall, the higher the degree of each data structure, the higher the performance, and since it is a read operation, the search performance of indexes which has the same degree tends to be similar. The CAWBT has a small degree, but overcomes the weakness through fast search algorithm and has almost the same performance as other indexes. In the case of fast-fair trees, linear search is used despite large orders, nevertheless the search performance is the highest.

5.6 Minimal Logging Performance

One minimal logging is comprised of three cache line flushes (status number writing, data writing, and status number clearing), three node reads (parent node, source node, new node), and CPU operations. The average logging time was calculated by measuring the sum of the minimal logging time when 0.5M KV pairs were inserted to the CAWBT. The total number of logging calls is 374,986, which was the same as the number of split operations because the logging code is called only when the split operation is executed. The total logging time was 36ms and the average logging time was 96ns. This is about four times the `clflush` time measured on average and is almost the same as the time set by design.

5.7 ID Table Re-Building Performance

Although the ID table is introduced to improve the I/O performance of the data structure, it is essential to recover the system when a system crash occurs due to the data existing in the DRAM. Figure 11 shows the time taken to recover the ID table by scanning the entire KV pair in an experiment using a workload that inserts 0.5M, 1M, 1.5M, and 2M KV pairs. Since the re-building algorithm is similar to sequential memory writing, thr result shows that the time linearly increases proportionally to the number of KV. It takes an average recovery time of 10ns per key, which is affordable.

5.8 Space Overhead

CAWBT occupies more memory because there is a mapping table in DRAM compared to other techniques. Figure 12 shows the space required per KV pair while storing 500,000

KVs. As a result, it takes up more space than wB+tree with higher order, but the diffrence is not notable.

6. Conclusion

In this paper, we introduced index structure called CAWBT which uses cache line sized atomic write. This index structure guarantees atomicity and consistency through cache line size atomic write and redo logging, and uses the minimum flushes to maximize the I/O performance of the NVM. This technique has been experimentally verified to operate normally, and the I/O performance is higher than general B+tree and the wB+tree, which is a novel index for NVM. Experimental results show that the performance of the proposed index structure is comparable to basic B+tree. It shows the performance of insert and search operation is 200% and 40% better than that of wB+tree, respectively.

Acknowledgments

This work is funded by Basic Research Lab Program (NRF, No. 2017R1A4A1015498) and ICT R&D program (IITP, R7117-16-0232). This work was also supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1G1A1100455).

References

[1] B.C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," ACM SIGARCH Computer Architecture News, pp.2–13, ACM, 2009.
[2] Panasonic, "The new microcontrollers with on-chip non-volatile memory reram," https://news.panasonic.com/jp/press/data/jn120515-1/jn120515-1.html, acceessed May 5 2012.
[3] J.F. Scott and C.A.P. de Araujo, "Ferroelectric memories," Science, vol.246, no.4936, pp.1400–1405, 1989.
[4] Y. Huai, "Spin-transfer torque mram (stt-mram): Challenges and prospects," AAPPS Bulletin, vol.18, no.6, pp.33–40, 2008.
[5] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon, "Frash: Exploiting storage class memory in hybrid file system for hierarchical storage," ACM Transactions on Storage (TOS), vol.6, no.1, p.3, 2010.
[6] J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, New York, NY, USA, pp.133–146, ACM, 2009.
[7] S.R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14, New York, NY, USA, pp.15:1–15:15, ACM, 2014.
[8] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," Proceedings of the Eleventh European Conference on Computer Systems, p.12, ACM, 2016.
[9] J. Coburn, A.M. Caulfield, A. Akel, L.M. Grupp, R.K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," ACM Sigplan Notices, vol.46, no.3, pp.105–118, 2011.
[10] H. Volos, A.J. Tack, and M.M. Swift, "Mnemosyne: Lightweight persistent memory," ACM SIGARCH Computer Architecture News, pp.91–104, ACM, 2011.
[11] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, "Software persistent memory," presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp.319–331, 2012.
[12] T. Hwang, J. Jung, and Y. Won, "Heapo: Heap-based persistent object store," ACM Transactions on Storage (TOS), vol.11, no.1, p.3, 2015.
[13] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," 2017 USENIX Annual Technical Conference (USENIX ATC 17), pp.349–362, 2017.
[14] S. Venkataraman, N. Tolia, P. Ranganathan, R.H. Campbell, et al., "Consistent and durable data structures for non-volatile byte-addressable memory.," FAST, pp.61–75, 2011.
[15] S.L. Graham, P.B. Kessler, and M.K. McKusick, "Gprof: A call graph execution profiler," SIGPLAN Not., vol.39, no.4, pp.49–57, April 2004.
[16] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," Proceedings of the VLDB Endowment, vol.8, no.7, pp.786–797, 2015.
[17] I. Moraru, D.G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems, TRIOS '13, New York, NY, USA, pp.1:1–1:17, ACM, 2013.
[18] D. Narayanan and O. Hodson, "Whole-system persistence," Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, New York, NY, USA, pp.401–410, ACM, 2012.
[19] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory," Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, New York, NY, USA, pp.371–386, ACM, 2016.
[20] H. Volos, A.J. Tack, and M.M. Swift, "Mnemosyne: Lightweight persistent memory," Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI, New York, NY, USA, pp.91–104, ACM, 2011.
[21] J. Yang, Q. Wei, C. Chen, C. Wang, K.L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems.," FAST, pp.167–181, 2015.
[22] D. Hwang, W.H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," 16th USENIX Conference on File and Storage Technologies (FAST 18), Oakland, CA, pp.187–200, 2018.
[23] T. Wang, J. Levandoski, and P.A. Larson, "Easy lock-free indexing in non-volatile memory," 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp.461–472, April 2018.
[24] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," Proceedings of the 26th Symposium on Operating Systems Principles, pp.460–477, ACM, 2017.
[25] Y. Mao, E. Kohler, and R.T. Morris, "Cache craftiness for fast multicore key-value storage," Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12, New York, NY, USA, pp.183–196, ACM, 2012.

Dokeun Lee received the B.S. degree in Electronic, Communication, Computer Engineering from Hanyang University, Korea in 2009. He is a Ph.D. candidate at Department of Computer Software at Hanyang University, Korea. His research interests include on NVM management system and filesystem for persistent memory.

Seongjin Lee is currently an Assistant Professor at Department of Aerospace and Software Engineering, Gyeongsang National University, South Gyeongsang Province, Korea. He did his BS and MS in Department of Electronics and Computer Engineering, Hanyang University, Seoul Korea in 2006 and 2008, respectively. He received his Ph.D. in Computer Engineering in the same university in 2015. Before joining Gyeongsang National University in 2017, he worked as PostDoc and assistant research professor at Hanyang University. His research interests include system performance, measurements, analysis, characterization, and classification.

Youjip Won is ICT Endowed Chair Professor at School of Electrical Engineering, KAIST. He did his BS and MS in Dept. of Computer Science, Seoul National University, Seoul,Korea in 1990 and 1992, respectively. He received his Ph.D. in Computer Science from University of Minnesota in 1997. He worked for Intel Corp. as Server Performance Analyst till 1999. From 1999 till 2019, he was with Dept. of Computer Science, Hanyang University, Seoul, Korea. His research interests include Filesystem, Storage System and Distributed System.