Ma RX, Wu F, Dong BR *et al.* Write-optimized B^+ tree index technology for persistent memory. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 36(5): 1037–1050 Sept. 2021. DOI 10.1007/s11390-021-1247-6

Write-Optimized B^+ Tree Index Technology for Persistent Memory

Rui-Xiang Ma[1,2,3,4], Fei Wu[1,2,3,4,*], *Senior Member*, *CCF*, *Member*, *IEEE*, Bu-Rong Dong[1,2,3,4]
Meng Zhang[1,2,3,4], Wei-Jun Li[5], *Senior Member*, *CCF*, *IEEE*, and Chang-Sheng Xie[1,2,3,4], *Member*, *IEEE*

[1] *Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430070, China*

[2] *Key Laboratory of Information Storage System, Ministry of Education of China, Wuhan 430070, China*

[3] *Engineering Research Center of Data Storage Systems and Technology, Huazhong University of Science and Technology Wuhan 430070, China*

[4] *School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430070, China*

[5] *Shenzhen DAPU Microelectronics Co., Ltd, Shenzhen 518116, China*

E-mail: {ruixiang, wufei, brong, zgmeng}@hust.edu.cn; wjli@semi.ac.cn; cs_xie@hust.edu.cn

Received December 31, 2020; accepted August 28, 2021.

Abstract Due to its low latency, byte-addressable, non-volatile, and high density, persistent memory (PM) is expected to be used to design a high-performance storage system. However, PM also has disadvantages such as limited endurance, thereby proposing challenges to traditional index technologies such as B^+ tree. B^+ tree is originally designed for dynamic random access memory (DRAM)-based or disk-based systems and has a large write amplification problem. The high write amplification is detrimental to a PM-based system. This paper proposes WO-tree, a write-optimized B^+ tree for PM. WO-tree adopts an unordered write mechanism for the leaf nodes, and the unordered write mechanism can reduce a large number of write operations caused by maintaining the entry order in the leaf nodes. When the leaf node is split, WO-tree performs the cache line flushing operation after all write operations are completed, which can reduce frequent data flushing operations. WO-tree adopts a partial logging mechanism and it only writes the log for the leaf node. The inner node recognizes the data inconsistency by the read operation and the data can be recovered using the leaf node information, thereby significantly reducing the logging overhead. Furthermore, WO-tree adopts a lock-free search for inner nodes, which reduces the locking overhead for concurrency operation. We evaluate WO-tree using the Yahoo! Cloud Serving Benchmark (YCSB) workloads. Compared with traditional B^+ tree, wB-tree, and Fast-Fair, the number of cache line flushes caused by WO-tree insertion operations is reduced by 84.7%, 22.2%, and 30.8%, respectively, and the execution time is reduced by 84.3%, 27.3%, and 44.7%, respectively.

Keywords persist memory, B^+ tree, write amplication, consistency, YCSB (Yahoo! Cloud Serving Benchmark)

1 Introduction

The explosive growth of data puts forward higher demands on database processing performance. In von Neumann's computer architecture, there is a huge difference in performance between CPU and the storage device. To alleviate the issue, high-performance databases reduce access to secondary storage devices by scaling up dynamic random access memory (DRAM) capacity. However, the process technology of DRAM has reached its limit, making it difficult to improve the DRAM capacity using scaling technology [1, 2]. Furthermore, a large number of studies have shown that the high energy consumption caused by the dynamic refresh operation of the DRAM has become the main bottleneck in improving system performance [3].

Regular Paper
Special Section of APPT 2021
This work was supported in part by the National Natural Science Foundation of China under Grant Nos. U1709220, U2001203, 61821003, 61872413, and 61902137, in part by the National Key Research and Development Program of China under Grant No. 2018YFB1003305, and in part by the Key-Area Research and Development Program of Guangdong Province of China under Grant No. 2019B010107001.

*Corresponding Author

The emergence of persistent memory (PM) can fundamentally address these problems. Its high density and low power consumption can overcome the bottleneck of DRAM, and PM is expected to replace DRAM as the main memory. Moreover, the non-volatility of PM can directly persist data on it. Consequently, PM can theoretically merge the memory layer with the storage layer. Therefore, databases using PM as the primary memory are an attractive design [4]. However, PM has problems with asymmetric read/write latency, limited write endurance, and data inconsistency, thereby posing challenges to current index technologies such as LSM-tree and B^+ tree. LSM-tree achieves better write performance by converting random write to sequential write and it is widely adopted by key-value store systems, including RocksDB① and HBase [5]. However, it incurs high read amplification. To alleviate the read amplification problem, LSM-tree performs periodic compaction to merge several sstables into a new sstable. Unfortunately, frequent compaction operation results in increased write amplification. Moreover, the compaction operation can take up device bandwidth and CPU cycles, resulting in a write stall. Some research confirms that compaction can become the new bottleneck in these new hardware devices [6]. Many LSM-tree variants have been proposed to adapt these hardware devices [7–10]. Unlike LSM-tree, B^+ tree supports fast read and range query. Therefore, B^+ is more suitable to read-intensive workloads and widely used in the relational database [11]. Since the difference between random and sequential access is smaller for these new hardware devices, the benefits of log-structured write are not obvious. Therefore, B^+ tree can be another better choice for the key-value store. In this paper, we focus on B^+ tree for PM. However, B^+ tree has poor write performance and high write amplification, and we aim to reduce the PM writes and improve the write throughput.

In this paper, we present WO-tree, a write-optimized B^+ tree index structure explicitly designed for PM. The main contributions of this paper are as follows.

• We propose a new B^+ tree variant called WO-tree, and WO-tree maintains the bitmap+slot array in the leaf node and a *next* pointer to the right sibling node.

• To improve the write performance, WO-tree uses a delayed flushing strategy during the inner node updating process. Moreover, the cache line flushing operation is performed after all write operations on the node are completed in WO-tree, which reduces the number of cache line flushes.

• WO-tree adopts a failure atomic recovery mechanism of partial logging and it only performs the logging operation on leaf nodes, which guarantees the data consistency and reduces logging overhead.

• We design an efficient lock-free search algorithm, which solves the concurrent performance degradation problem caused by the locking operation.

• The native B^+ tree, wB-Tree, Fast-Fair, and WO-tree are implemented and compared. Our results show WO-tree achieves superior performance on YCSB workloads.

The rest of this paper is organized as follows. In Section 2, we briefly introduce PM, B^+ tree, and motivation. The related work is discussed in Section 3. Section 4 describes the proposed design and implementation in detail. The experiment and the evaluation are presented in Section 5. Finally, the conclusions are described in Section 6.

2 Background and Motivation

2.1 Persistent Memory

In recent years, PMs such as Phase Change Memory (PCM) [12, 13], Resistive Random Access Memory (RRAM) [14] and Spin-transfer Torque Random Access Memory (STT-RAM) [15], have been extensively researched in the industry and academia. Recently, Intel Optane DC persistent memory② has been commercially available on the market. PM has the characteristics of byte-addressable, which allows it to be directly mounted on the memory bus and accessed via the load/store instructions. PM has very low read/write latency, high density, and low power consumption, which promises to replace DRAM as the main memory. In addition, PM is non-volatile and can be used as a persistent storage device. Table 1 presents the characteristics of various memory technologies.

2.2 Challenges for Persistent Memory

PM has become a research hotspot in academia and industry in recent years. However, PM has the problems of asymmetric read and write latency, limited endurance, and data inconsistency. Therefore, the emergence of PM has brought opportunities to the develop-

①https://rocksdb.org, Sept. 2021.

②https://investors.micron.com/static-files/7b934cfe-139c-4a6f-93e5-b86240642351, Sept. 2021.

Table 1. Characteristics of Different Memory Technologies

Feature	Read Latency (ns)	Write Latency (ns)	Endurance	Dynamic Power	Non-Volatility
DRAM	10	10	$> 10^{15}$	High	No
STT-RAM	2–20	5–50	10^{12}–10^{15}	Low	Yes
PCM	150	450	10^{7}–10^{8}	Low	Yes
RRAM	10	50	10^{8}–10^{10}	Low	Yes

ment of computer architecture, as well as challenges to today's software technology. When migrating conventional technologies to a PM-based system, the following should be taken into account.

Mfence and Cflush. To improve system performance, CPU reorders memory writes when writing back to PM. Consequently, the order of the cache line write-back is uncertain, which makes it difficult to correctly recover the data when the system crashes. To address this problem, the instructions mfence and cflush are used to ensure the order of cache line flushing. However, a large number of studies have proved that mfence and cflush operations cause a large write overhead, resulting in a decrease in system performance [11, 16–18].

Double Write. The atomic write granularity supported by CPU is 8 bytes while the size of a cache line is 64 bytes. The granularity mismatch between the cache line and the atomic write can cause partial data updates when the system crashes. Due to the non-volatility of PM, the data can be inconsistent when the system crashes. In a disk-based system, Write-Ahead Logging (WAL) [16–18] is commonly used to ensure data consistency. However, WAL causes a double write problem. For DRAMs with balanced read/write latency and unlimited write endurance, the system performance will not be significantly impaired. For PM, the logging scheme not only causes a lot of clflush and mfence operations, but also makes rapid wear-out of PM devices.

Lock Overhead. In the multithread mode, there is a locking mechanism for accessing this shared data. The locking mechanism significantly impacts the concurrency performance of the database system. Moreover, the acquisition and the release of the lock also cause additional write operations.

2.3 Write Amplification in B^+ Tree

B^+ tree is an index structure that is designed for a disk-based storage system [19]. B^+ tree is a self-balancing search tree and has the characteristics of high node fan-out, balanced tree height, and dynamic adjustment of the node size [20]. B^+ tree has excellent data access performance and is widely used in relational databases such as Mysql. In B^+ tree, the node is divided into two types: inner node and leaf node, and their layout is shown in Fig.1. The inner node stores the index information pointing to the next level node and does not store the actual data. The leaf node stores the information of inner nodes and entries. In B^+ tree of order m, each inner node contains m keys and m pointers that point to the children. Each leaf node contains n key-value pairs and a pointer pointing to the sibling node, which forms a linked list.

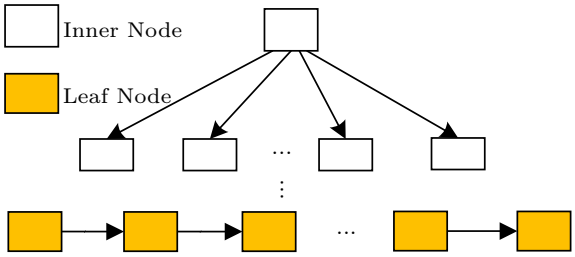

Fig.1. Layout of B^+ tree [19].

In the B^+ tree, all entries are stored in the order of keys. To keep the order of the entries inside the node, the data reordering operation is frequently triggered. We suppose that the key of the existing entries is 2, 4, 5, 7, 9 respectively. As presented in Fig.2, *num* represents the number of valid entries and the node is in the initial state 1. When a new entry $(6, value)$ is inserted in the node, these valid entries need to be sorted according to the key. A suitable insertion position can be found with binary search technology. As shown in states 2, 3, 4, and 5, the other entries shift right after finding a suitable location. *num* is updated to 6 when the new entry is inserted. For insertion operation, the number of shifts is 0 if the key of the inserted entry is larger than the maximum key of the current node. But if the key of the inserted entry is less than the minimum value of the current node, then all entries in the node move to the right. Therefore, there are $N = \frac{m+1}{2}$ shift operations in B^+ tree (m is the number of entries in the inner node). For the deletion operation, these entries will be shifted to the left, causing the same problem.

The system failure may occur during the above process, which causes data inconsistency. To ensure system crash recovery, a WAL operation is also performed prior to each write operation.

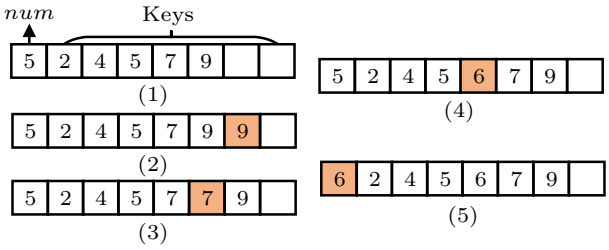

Fig.2. Example of the insertion operation for B^+ tree.

The write amplification caused by reordering and logging makes PM wear out quickly and seriously degrades the index performance. Furthermore, to ensure the write order, write operations in PM must call mfence and clflush instructions. The mfence and clflush instructions are the main cause of the index performance degradation. Therefore, it is necessary to design a PM-based B^+ tree index structure that can take into account the characteristics of PM.

3 Related Work

Nowadays, researchers have proposed some optimization solutions for the application of the B^+ tree in PM.

Venkataraman *et al.*[16] proposed CDDSs, which are designed for PM. CDDSs use version control to allow atomic updates without logging. However, CDDS needs to perform frequent flush operations during each update operation, which causes a lot of overhead.

Yang *et al.*[17] proposed NV-Tree. NV-Tree uses an append-only write scheme and these entries in leaf nodes are unordered. However, a binary search cannot be used in NV-Tree, which affects search performance.

Oukid *et al.*[18] proposed FP-Tree, which stores inner nodes in DRAM and leaf nodes in PCM. FP-Tree only performs persistent operations on leaf nodes. In addition, FP-Tree reduces access to PM through fingerprint. As the inner nodes are in DRAM, frequent data reconstruction is required when the system fails.

Chen and Jin[11] proposed wB-tree. The leaf nodes are unordered and use an append write scheme similar to NV-Tree. To improve the search performance, wB-tree uses a slot structure to guarantee entry order. However, wB-tree does not consider the lock overhead of multi-threaded access to maintain consistency.

Ni *et al.*[21] proposed a Bp^+ tree based on a hybrid memory of DRAM and PCM. The Bp^+ tree uses a prediction mechanism to determine the allocation of PM or DRAM. However, the total data volume needs to be known in advance. In addition, the accuracy of the prediction result cannot be guaranteed. At the same time, due to the volatility of DRAM, it is difficult to realize instant recovery in the event of an unexpected system failure.

Hwang *et al.*[22] proposed FAST-Fair. FAST-Fair can identify the inconsistent entry, which can avoid expensive logging overhead. However, the problem of heavy write amplification caused by reordering is still unsolved.

The above studies achieve excellent performance. However, they focus only on some aspects of the above challenges. Based on existing work, this paper proposes a PM-based write optimization B^+ tree index structure, which solves the write amplification caused by reordering, the high logging overhead for failure recovery, and the locking overhead for multi-thread mode.

4 WO-Tree

This section presents the design of WO-tree and its implementation in detail.

4.1 WO-Tree Node Layout

To address the problems in PM, we propose WO-tree, a write-optimized B^+ tree index structure for PM. Fig.3 shows the layout of the leaf node and the inner node of the WO-tree.

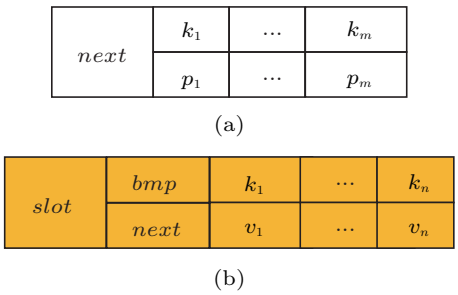

Fig.3. Node layout of WO-tree. (a) Inner node. (b) Leaf node.

In WO-tree, the data structure of bitmap *bmp*+slot array *slot*[11,23] is maintained in the leaf node, which aims to improve the overall system performance. The design of insertion, deletion, and split operation for leaf nodes is discussed in detail in Subsection 4.2. In the inner node, WO-tree maintains a *next* pointer to the

right sibling node, which is similar to the leaf node. It aims to address the data loss problem caused by the node split operation. The insertion, deletion, and split operations for an inner node are discussed in detail in Subsection 4.3.

4.2 Leaf Node

In the native B^+ tree, entries are stored in order. To maintain the characteristic, the insertion operation moves half of the entries of a node on average. Since there is no dependency between the shift operations of the leaf node, CPU can write back the data to PM out of order. To ensure the data consistency, write operations have to call a large number of clflush and mfence instructions. To reduce the number of cache line flushes caused by the data reordering, WO-tree adopts the unsorted write scheme for the insertion operation in the leaf node.

In WO-tree, *bmp* records the usage of the node. The valid bit corresponds to 1 while the invalid bit corresponds to 0. The entry is directly appended in the first available empty position of the node when inserting a new entry. WO-tree resets the corresponding position in the bitmap to 0 when deleting an entry, and it can be directly overwritten in the position whose bit is 0. The unsorted write scheme makes the binary search method fail, thereby decreasing search performance. To address the problem, WO-tree implements the binary search mechanism with the slot array structure *slot*. *slot*[0] records the number of valid entries of the node, and the following *slot*[1]–*slot*[n] store the offset of the corresponding entries in the order of the key, where n represents the number of valid entries in the leaf node. In WO-tree, the leaf node consists of the counter, bitmap, and slot array, which causes the extra space amplification. For example, we suppose the node size is 64 cache lines, the sizes of key and value are both 8 bytes, and m represents the number of the entry in this node, $m < \frac{64\times64}{8+8} = 256$. Therefore, 1 byte can be used to represent the offset of entry in the slot array, *slot* takes m bytes, the bitmap needs $\frac{m}{8}$ bytes, the counter takes 1 byte to represent the number of the valid entry, and $m \times (8+8)$ byte is used to represent the entry. The space amplification caused by the bitmap and slot array becomes smaller when the value becomes bigger. It is because m decreases with the increase of key size or value size. Therefore, WO-tree is very suitable for a larger value or larger keys. The benefit of WO-tree is introduced in Subsection 4.2.1.

4.2.1 *Insertion Operation*

When performing an insertion operation, the entry is directly appended in the leaf node. WO-tree compares the new entry with other entries and updates the bitmap and slot array. Algorithm 1 is the pseudo-code of the insertion operation in the leaf node.

Algorithm 1. Insertion Operation in Leaf Node

Input: (*key*, *value*)

Output: NULL

/*Find the leaf node where the key is stored*/

1: TraverseLeaf(*key*, *parent*, *leaf*)

/*Find free location*/

2: $p \leftarrow$ findposwithbitmap()

/*Insert the new entry into the position*/

3: *key* $\leftarrow$ data[*p*].key, *value* $\leftarrow$ data[*p*].value

4: cflush(*leaf*.data[*p*]); mfence

/*Update the slot array and bitmap*/

5: BitmapUpdate(*leaf*, *key*, *p*)

6: SlotUpdate(*leaf*, *key*, *p*)

7: cflush(bitmap); mfence

8: cflush(slot); mfence

As shown in Fig.4, a new entry (6, *value*) is inserted into the leaf node. The initial state is in state 1. When inserting the entry, WO-tree first finds the valid position using the bitmap. The new entry is then inserted into the position and the result is shown in state 2. We set the corresponding bit to 1 in *bmp*. The slot array is updated into 6, 0, 1, 2, 5, 3, 4, which is shown in state 3. When performing an insertion operation in the leaf node, only the newly inserted entry, slot array,

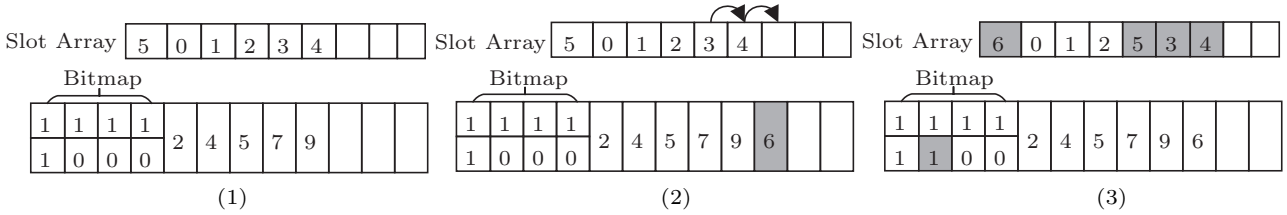

Fig.4. Example of an insertion operation in a leaf node.

and bitmap are flushed, which greatly reduces the write operation for the PM. As mentioned in Subsection 2.3, there are $N = \frac{n+1}{2}$ cache line flushing operations in the native B^+ tree. Compared with the native B^+ tree, the number of cache line flushes in WO-tree is decreased from $O(n)$ to $O(1)$.

4.2.2 Search Operation

Due to the unsorted write mechanism, the leaf nodes can only be traversed to find the key, which degrades the search performance. As the slot array of WO-tree records the offset ordered by key in the leaf node, the binary search can be implemented using the slot array. The search operation is represented in Algorithm 2 in detail.

Algorithm 2. Search in Leaf Node

Input: *key*
Output: *pos*

```
1:  l ← 1, r ← slot[0] + 1;
2:  while l ⩽ r do
3:    mid ← (l+r)/2;
4:    if key ⩽ data[slot[mid]].key then
5:      r ← mid − 1;
6:    else
        l ← mid + 1;
7:    end if
8:  end while
9:  if r ⩽ slot[0] and data[slot[r]].key == key then
10:   return r;
11: end if
```

4.2.3 Deletion Operation

In WO-tree, the deletion operation is similar to the insertion operation. WO-tree first finds the offset using the above search algorithm and sets the corresponding position in the bitmap to 0. WO-tree then updates the slot array and eventually persists the bitmap and slot array.

As shown in Fig.5, we suppose that entry(5, *value*) is deleted in the node. WO-tree first sets the bit position in the bitmap to 0 and then removes its offset in the slot array. *slot*[0] is finally updated into 5. Therefore, the number of cache line flushes is $O(1)$.

4.2.4 Split Operation

Supposing the available entries in the leaf node are full, it can cause the node to split when a new entry is inserted into this node. For split operation, these entries in the node are first divided into two parts and half of the entries are then inserted into the newly created node. The new node information is finally updated to the parent node.

In WO-tree, the entry is unsorted in each leaf node while the node is sorted in the tree structure. Therefore, the entries in the split node must be reordered during the split operation. These entries are accessed in order using the slot array and half of the entries are migrated to the new node. The sibling pointer of the node points to the new node and the new leaf node information is updated to the parent node. If the parent node is empty, this means that the current node is the root node and the height of the tree needs to be increased to store this index information. In addition, to avoid a large number of cache line flushes caused by each write operation, WO-tree performs the data flushing operation after all write operations are completed, which can reduce frequent data flushing operations. The detailed design is presented in Algorithm 3.

Supposing the leaf node can store 16 entries and the key of these entries ranges from 0 to 15. The node is split when the node is full, and the result is represented in Fig.6(a). To ensure the order among the nodes, the original node cannot be directly divided into two parts. We need to guarantee that the keys of its left sibling node are larger than the keys of the node. We first move the entry using the slot array, and the entries of the leaf nodes are stored in two nodes in order. The slot arrays and bitmap of the two nodes are updated synchronously. The result after splitting is shown in Fig.6(b).

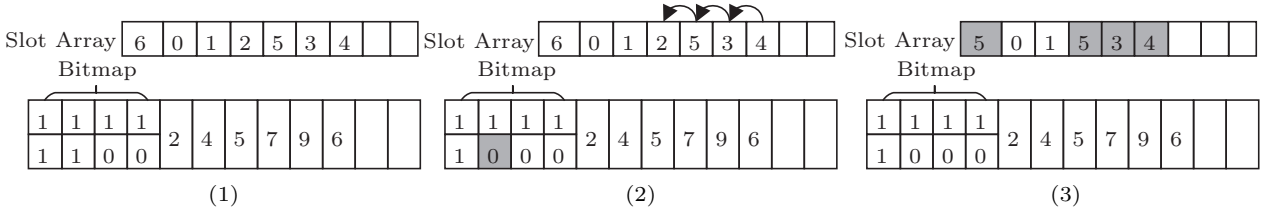

Fig.5. Example of a deletion operation in a leaf node.

Algorithm 3. Split Operation in Leaf Node

Input: node

Output: newnode

/*Migration entry to the new node*/

1: $size \longleftarrow slot[0]$, LeafNode newnode[$size$]
2: **for** $i \longleftarrow \frac{size}{2}$; $i \leqslant size$; $i++$ **do**
3: newnode[i] $\longleftarrow$ data[$slot[i]$]
4: **end for**

/*Adds a pointer to the sibling node*/

5: node.next $\longleftarrow$ newnode.next
6: newnode $\longleftarrow$ node.next
7: newnode.children_number $\longleftarrow \frac{size}{2}$
8: node.children_number $\longleftarrow \frac{size}{2}$
9: cflush(newnode); mfence
10: cflush(node); mfence
11: return newnode

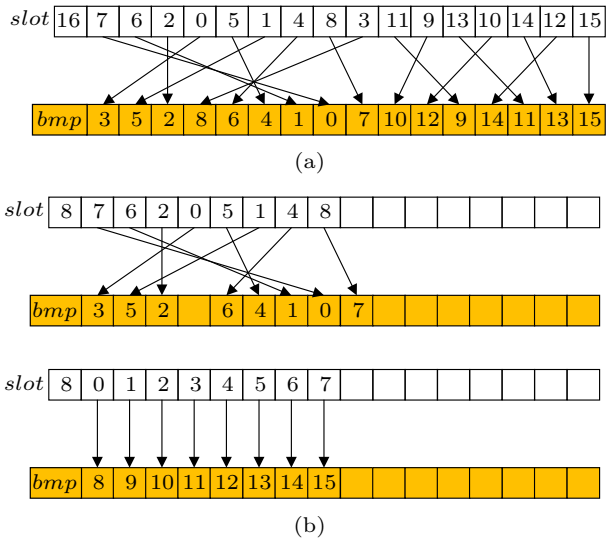

Fig.6. Leaf node. (a) Original node. (b) Two nodes after leaf node split.

4.3 Inner Node

Taking into account the problem of search performance, WO-tree does not use the unsorted write mechanism on the inner nodes.

4.3.1 Insertion Operation

To reduce the number of cache line flushes in inner nodes, WO-tree adopts a reverse movement strategy, similar to the Fast-Fair algorithm [14]. By adding the dependency between entries, CPU reordering operations can be prohibited. The problem of data consistency can be solved as follows: when the index information of a new node needs to be inserted into an inner node, the entry is accessed from the right to the left. Before finding a suitable position, these entries shift right in the node. Furthermore, WO-tree moves the entry only after the previous items have been successfully inserted. The data consistency problem can occur if the system crashes. However, data inconsistency can be identified by reading the left and the right pointers. If the left and the right pointers are the same, we ignore the entry information and continue to access backward to find the correct entry.

To reduce the number of cache line flushes, WO-tree adopts a delayed flush strategy. The cacheline flushing operation is triggered only when the data moves from one cache line to another adjacent cache line. This is because the order of cache line flush is undetermined if the cflush and mfence instructions are not called when moving across cache lines. It is detrimental to recover the data when a system crashes. Supposing that the size of an entry is 16 bytes, a cache line can store 4 entries. As shown in Fig.7, entries 0–3 are stored in cache line 1, and entries 4–7 are stored in cache line 2. Only after ptr3 moves from cache line 1 to cache line 2, the data flushing operation is triggered. Frequent flush operations can be reduced using the delay flush scheme. The number of cache line flushes is reduced from half of the original nodes to the number of dirty cache lines, that is, $\frac{m}{2}$ (m is the number of entries contained in a node) to $\frac{n}{2}$ (n is the number of cache lines spanned by a node). Assuming that a node contains 64 entries, each entry contains an 8-byte key and an 8-byte pointer. Therefore, the node spans 16 cache lines. When inserting a new entry into the node, the number of cache line flushes is reduced from $\frac{64}{2} = 32$ to $\frac{16}{2} = 8$. The number of cache line flushes is only $\frac{1}{4}$ of the original.

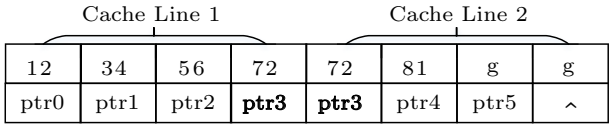

Fig.7. Data flushing is triggered when the data move from one cache line to another adjacent cache line.

4.3.2 Split Operation

When the leaf node is split, the index information of the new node needs to be updated to that of the parent node, which may cause the split operation of the parent node. The node then passes this information to the upper parent node in turn until it reaches the root node. If a failure occurs before the index information reaches the upper node, the newly-generated information can

be lost. However, the WAL operation is not used for inner nodes and the data are not recovered from the log.

To address the problem, WO-tree maintains a pointer *next* to the sibling node in the inner node. *next* points to the new node before deleting the original entry of the split node. Even if an unexpected failure occurs in the split operation, the new node information can be accessed using the pointer *next*, thereby ensuring data integrity. In WO-tree, a split operation is considered a success when the key of the entry on the right is greater than the key of the entry on the left, and the pointer is unique. Thus, data consistency can be guaranteed when the node is split. The detailed process is presented in Algorithm 4, which is similar to the split operation in the leaf node.

Algorithm 4. Split Operation in Inner Node

Input: node

Output: newnode

/* Migrate entry to the new node*/

1: InnerNode newnode[$size$]
2: **for** $i \longleftarrow 1; i \leqslant size; i++$ **do**
3: newnode[i] $\longleftarrow$ data[i]
4: **end for**

/*Adds a pointer to the sibling node*/

5: node.next $\longleftarrow$ newnode.next
6: newnode $\longleftarrow$ node.next
7: newnode.children_number $\longleftarrow \frac{size}{2}$
8: node.children_number $\longleftarrow \frac{size}{2}$
9: cflush(newnode); mfence
10: cflush(node); mfence
11: return newnode

4.3.3 Deletion Operation

As the inner node uses a pointer to the sibling node, a linked list structure is created between the nodes of the same level. Furthermore, each inner node is sorted. Therefore, the inner node is similar to the leaf node in the native B^+ tree and the deleting operation is the same as the leaf node operation in the native B^+ tree.

4.3.4 Failure Atomic Recovery Mechanism

To guarantee the data consistency caused by system crashes, the WAL operations are performed on both leaf nodes and inner nodes for the disk-based and DRAM-based index structures. However, the WAL scheme makes a great impact on system performance. To alleviate this problem, WO-tree uses a partial WAL mechanism to reduce logging overhead. WO-tree only implements the WAL scheme for the leaf node. As shown in Fig.8, the log is first written, and then the leaf node is updated. In the B^+ tree, the leaf nodes also store the information from inner nodes. Therefore, the information from the inner node can be recovered through the leaf nodes. According to the introduction of Subsection 4.3.1, inner nodes can recognize data inconsistency through reading operations. Once an inconsistent state is detected, the data can be restored via the leaf node. In the case that node information cannot be successfully updated to its parent node during the split operation, the related data recovery method has been introduced in detail in Subsection 4.3.2.

When rebuilding the index tree, WO-tree first identifies whether the inner node is in a consistent state, and builds the index tree frame. If the entry is in an

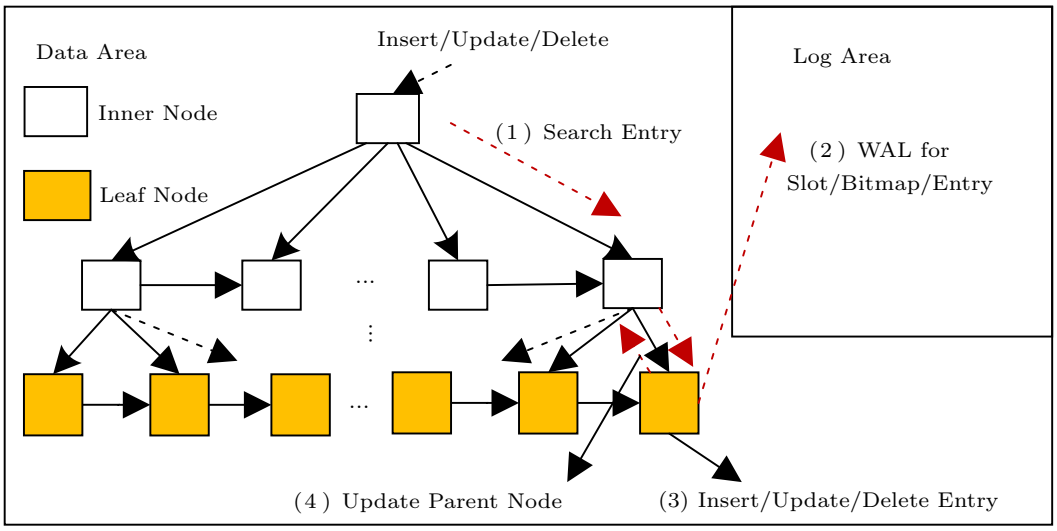

Fig.8. WAL for leaf nodes.

inconsistent state, it is ignored directly. We then read other entries in a consistent state until all correct entries in the inner node are completed. Finally, the leaf node is read from the log area and these entries are inserted into the constructed tree frame. As PM has lower latency, WO-tree can achieve instant recovery. This detailed process is shown in Fig.9.

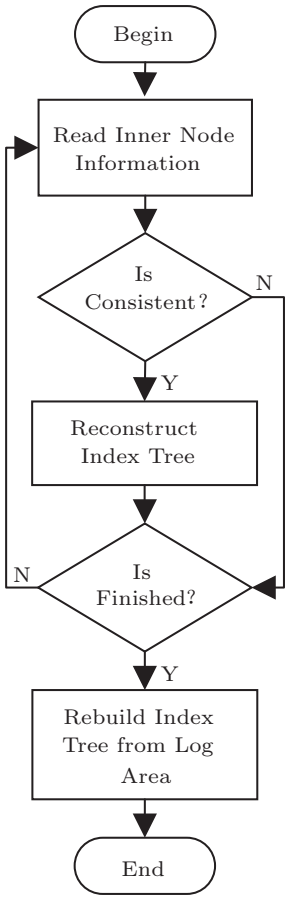

Fig.9. Rebuild flow chart of WO-tree.

4.3.5 *Lock-Free Search*

In the multithread mode, a locking mechanism is typically used to ensure data consistency. However, the locking mechanism can result in a decrease in concurrent performance. Since system crash is rare, WO-tree uses the lock-free search scheme for inner nodes. However, there are phantom and dirty read problems in this lock-free search scheme. As introduced in Subsection 4.3.1, the data consistency can be detected in the inner node. Thus, the read thread can identify the inconsistent data and the data can be recovered by the failure atomic recovery mechanism above. In the leaf nodes, to ensure the consistency of critical data, we apply a read lock in the leaf node.

5 Experiment and Evaluation

5.1 Experimental Setup

In this paper, we run experiments on Ubuntu 18.04 (5.3.0-46-generic) with Intel Xeon E5-2450 (8 cores/16 threads 2.2 GHz), 16 GB of DRAM, as shown in Table 2. PM is emulated by DRAM using a hardware emulator [24]. DRAM of 8 GB is first allocated to emulate PM, and we then mount a DAX-enabled ext4 file system on it. Since the read and the write latency of PM are unbalanced, we inject an extra write delay for each write operation, which is similar to Quartz [25]. Up to 400 ns is taken as the default extra latency between PM read and PM write operation.

Table 2. Experiment Platform

Parameter	Value
OS	Ubuntu18.04
Linux version	5.3.046-generic
CPU	E5-4603 8 cores/16 threads 2.2 GHz
PM	8 GB RAM

5.2 Workload

The workloads used in this experiment are YCSB (Yahoo! Cloud Serving Benchmark) [26]. The experiments compare the existing index structures with WO-tree. In the experiment, each operation is performed 10 times and the average result is the final result.

5.3 Evaluation and Analysis

In this experiment, the sizes of the key and the value are both 8 bytes and the size of the node is 4 cache lines, 8 cache lines, 16 cache lines, and 32 cache lines respectively. This subsection compares WO-tree with the native B^+ tree, Fast-Fair, and wB-Tree in terms of insertion, deletion, and search performance.

5.3.1 *Insertion*

For the insertion operation, we mainly count the number of cache line flushes (including cflush and mfence) of PM. We also compare the execution time caused by insertion operations under various PM write delays.

In this experiment, the number of inserted entries is 300 000. The experimental results are presented in Fig.10(a). WO-tree gets the best result when the node size exceeds 8 cache lines. This is because the small node size can cause frequent splitting operations, and

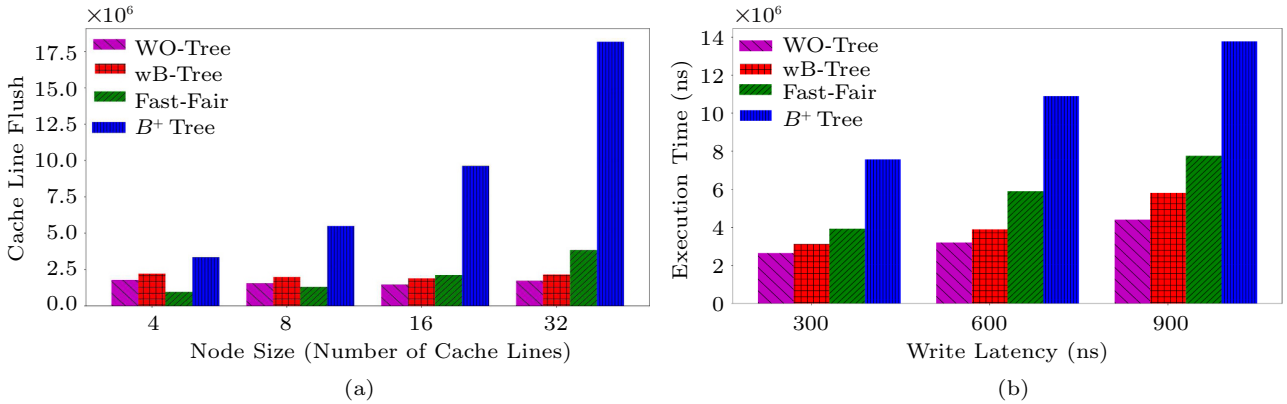

Fig.10. Insertion operation for 300 000 entries. (a) Number of cache line flushes. (b) Excution time with varied write latency.

the splitting operation leads to a lot of update operations on the slot array and bitmap. The number of entries increases when the node size increases. Therefore, the number of split operations can be significantly reduced, and the number of flushes for wB-tree and WO-tree is significantly reduced. However, when the node size exceeds 16 cache lines, the number of cache line flushes increases. This is because it causes frequent updates for slot arrays when the node is too large. To further reduce the number of cache line flushes caused by the splitting operation, the data flush operation is triggered only after all the write operations are completed during the splitting process. Compared with the native B^+ tree, wB-tree, and Fast-Fair, the number of cache line flushes in WO-tree is reduced by 84.7%, 22.2%, and 30.8% respectively when the node size is 16 cache lines.

To evaluate the impact of write delay on the write performance, we test the execution time under various PM write latency. In this experiment, the node size is 16 cache lines, and the extra write delay of PM varies between 300 ns and 600 ns. Fig.10(b) shows the experimental results. We can find that WO-tree has the best performance under various write delays. The execution time gap between the four index structures gradually increases as write latency increases. Therefore, the write latency affects the performance of the index structure. Compared with native B^+ tree, wB-tree, and Fast-Fair, the execution time of WO-tree is reduced by 71%, 22.5%, and 43.3% respectively when the write delay is 300 ns.

5.3.2 *Deletion*

To perform the deletion experiment, we first insert 1 000 000 entries, and then delete 300 000 entries. Fig.11(a) shows the number of cache line flushes. The results show the native B^+ tree performs the worst under different node sizes. This is because the shift operation is frequently triggered, thereby causing a large number of cache line flushes. WO-tree and Fast-Fair use the delayed flushing strategy, which dramatically reduces the number of cache line flushes. WO-tree and wB-tree only reset the corresponding bit in the bitmap to 0 and update the slot array. Compared with the na-

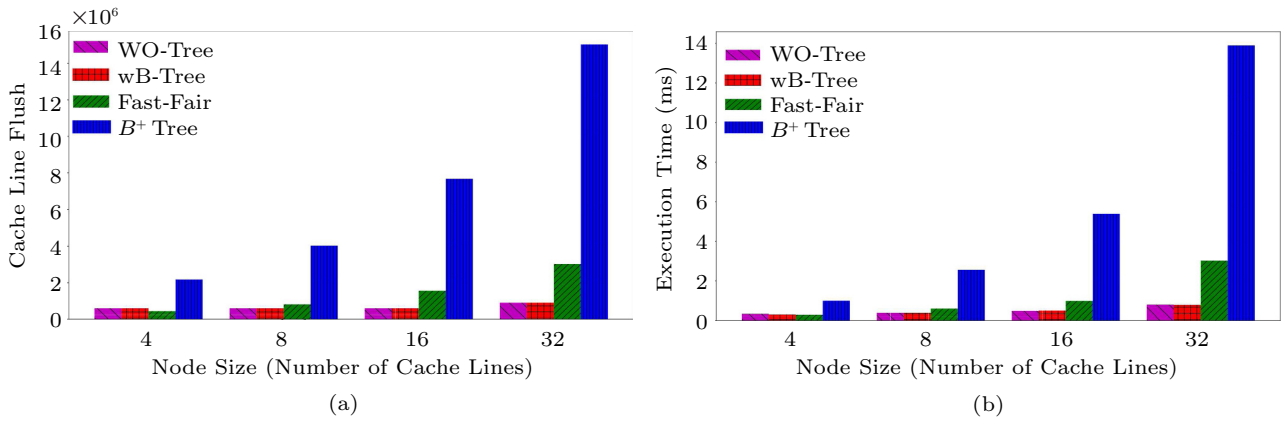

Fig.11. Deletion operation for 200 000 entries. (a) Number of cache line flushes. (b) Execution time.

tive B^+ tree and Fast-Fair algorithm, WO-tree reduces the number of flushes by 99.2% and 61.4% respectively when the leaf node size is 16 cache lines. Fig.11(b) shows the execution time of the 300 000 deletion operation. Compared with the native B^+ tree, wB-tree, and Fast-Fair, the execution time is reduced by 91%, 2.6%, and 50.5% respectively when the node size is 16 cache lines.

5.3.3 *Search*

In the search experiment, we record the execution time for the four indexing structures. Since no write operation occurs during the search operation, the number of cache line flushes is taken into account. We first insert 1 000 000 entries and then record the execution time of 300 000 search operations.

Fig.12 shows the experimental results. It can be found that the search latency of the native B^+ tree takes the least time when the node size is the same, and Fast-Fair takes the most time. This is because the entry can be directly queried using the binary search for the native B^+ tree, and wB-tree and WO-tree must use slot array to implement the binary search method. However, the slot array needs to be first accessed, which brings extra access latency. Fast-Fair needs to traverse the inner node, which leads to a decrease in search performance. Compared with the native B^+ tree and wB-tree, the execution time of the WO-tree increases by 2.74% and 1.64% respectively when the node size is 16 cache lines. However, the lock-free search avoids the locking overhead, and the search performance can be improved for concurrent operation.

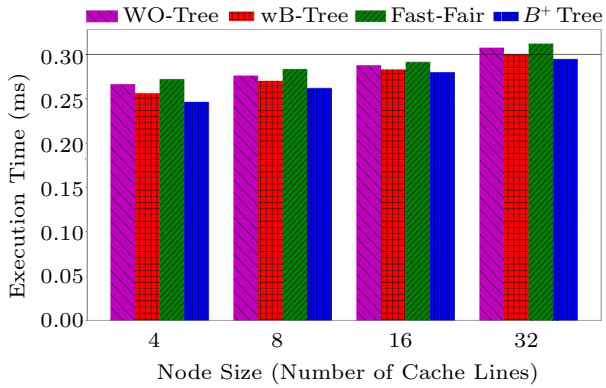

Fig.12. Search execution time with a varied node size.

5.3.4 *Concurrent*

This subsection compares the concurrent performance of WO-tree, native B^+ tree, Fast-Fair, and wB-tree in terms of insertion, deletion, and search operation. In this experiment, the node size is set to 16 cache lines.

For the insertion operation, the number of insertion entries is 300 000, and the number of threads is 1, 2, and 4, respectively. Fig.13(a) presents the experimental results of the four index structures. Compared with the native B^+ tree, the execution time of WO-tree is reduced by 83.5%, 84.3%, and 87.2% under various thread counts, respectively. Compared with wB-tree, the execution time of the WO-tree has been reduced by 17.6%, 27.3%, and 27.1% under various thread counts, respectively. Compared with Fast-Fair, the execution time of WO-tree has been reduced by 40.6%, 44.7%, and 46.2% under various thread counts, respectively.

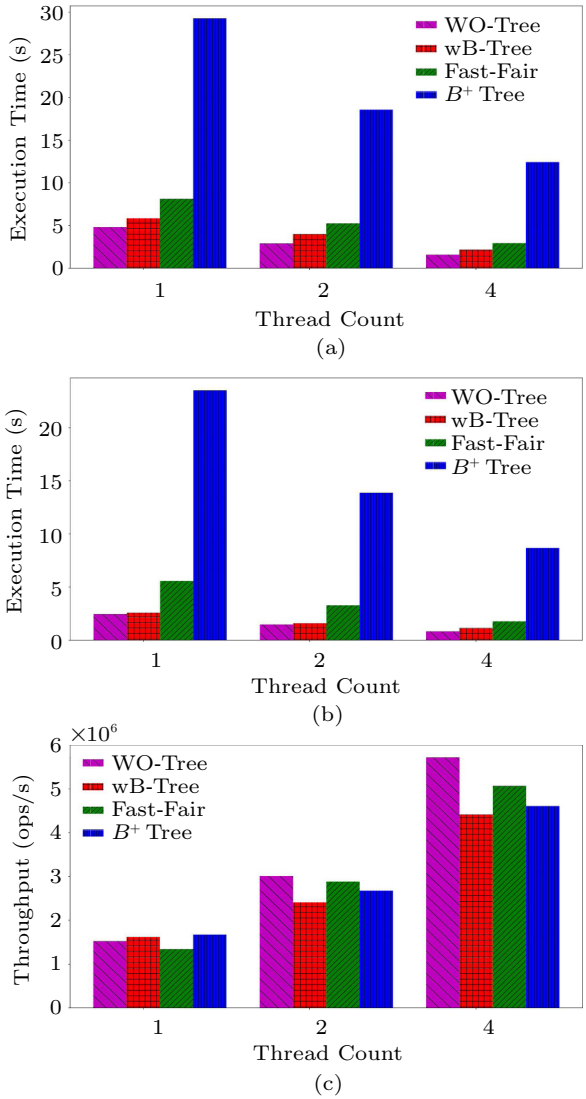

Fig. 13. Concurrency performance with different numbers of threads. (a) Insertion execution time. (b) Deletion execution time. (c) Throughput.

For the deletion operation, 1 000 000 entries are first inserted, 300 000 entries are then deleted and the execution time of the deletion operation is counted. Fig.13(b) presents the results of the above experiment. Compared with the native B^+ tree, the execution time of WO-tree is reduced by 89.6%, 89.4%, and 90.5% under various thread counts, respectively. Compared with the wB-tree, the execution time of the WO-tree is reduced by 4.9%, 7.5%, and 28.1% under various thread counts, respectively. Compared with Fast-Fair, the execution time of the WO-tree has been reduced by 56%, 55%, and 53.7% under various thread counts, respectively.

For the search operation, we first insert 1 000 000 entries and then calculate the throughput (the number of queries per second) for 300 000 query operations, and Fig.13(c) shows the above experimental results. Compared with the native B^+ tree when the number of threads is 1, the throughput of WO-tree is reduced by 9%. As the number of threads increases, the throughput of WO-tree has increased by 12.5% and 24.1 % compared with the native B^+ tree, respectively. Compared with wB-tree, the throughput of WO-tree is reduced by 5.9% when the number of threads is 1. This is because WO-tree needs to perform data consistent validation. As the number of threads increases, the throughput of the WO-tree increases by 25% and 29.5%, respectively. Compared with Fast-Fair, the throughput of WO-tree increases by 13.6%, 4.3%, and 12.8% under a different number of threads, respectively. Due to the lock-free scheme adopted by WO-tree for inner nodes, the search performance can be greatly improved.

5.3.5 *Mixed Workload*

This subsection compares the performance of WO-tree, the native B^+ tree, FAST-FAIR, and wB-tree on mixed workloads. In this experiment, 1 000 000 operations are generated. The read and write ratios are 30%/70%, 50%/50%, and 70%/30% respectively, and the number of threads is set to 4.

Fig.14 presents the experimental results of the four index structures under different mixed workloads. Compared with native B^+ tree, wB-tree, and FAST-FAIR, the execution time of WO-tree is reduced by 72.6%, 11.5%, and 26.4% respectively when the read-write ratio of the load is 30%/70%. The execution time of WO-tree is reduced by 73.3%, 17.7%, and 30.8% respectively when the read-write ratio is 50%/50%. The execution time of WO-tree is reduced by 60.6%, 34.2%, and 19.2% respectively when the read-write ratio is 70%/30%. It can be found that WO-tree has the most significant performance improvement when the proportion of insertion operations is maximum. This is because WO-tree can significantly reduce the number of cache line flushes. As the proportion of query operations increases, WO-tree always shows better performance. This is because WO-tree uses the lock-free search in the inner nodes, which reduces the number of cache line flushes.

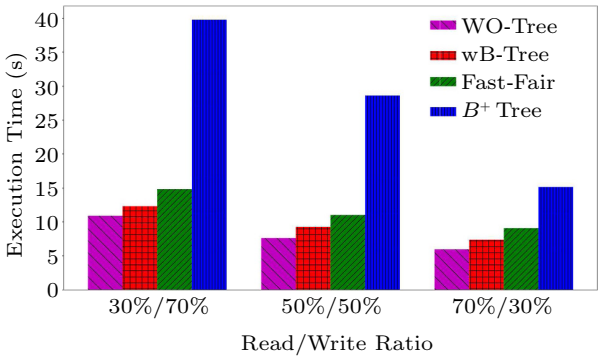

Fig.14. Execution time of mixed workloads.

6 Conclusions

This paper proposed a PM-based write-optimized B^+ tree, WO-tree, which is mainly to reduce the expensive write operations in PM. WO-tree addresses the write amplification problem using an unsorted write scheme. WO-tree applies a delayed flushing strategy to reduce the frequent cache line flushing operations, adopts a partial logging mechanism that only performs logging for the leaf nodes, and uses a lock-free search scheme to reduce locking overhead in inner nodes. Experimental results showed that WO-tree can achieve a higher insertion, deletion, and concurrent performance than both wB-tree and Fast-Fair on YCSB workloads.

References

[1] Mueller W, Aichmayr G, Bergner W *et al.* Challenges for the DRAM cell scaling to 40nm. In *Proc. IEEE International Electron Devices Meeting*, December 2005, pp.336-339. DOI: 10.1109/IEDM.2005.1609344.

[2] Mandelman A J, Dennard H R, Bronner B G *et al.* Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM Journal of Research and Development*, 2002, 46(2.3): 187-212. DOI: 10.1147/rd.462.0187.

[3] Freitas R, Wilcke W. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 2008, 52(4.5): 439-447. DOI: 10.1147/rd.524.0439.

[4] Arulraj J, Pavlo A, Dulloor S. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proc. the 2015 ACM SIGMOD International Conference on Management of Data*, May 31–June 4, 2015, pp.707-722. DOI: 10.1145/2723372.2749441.

[5] Harter T, Borthakur D, Dong S *et al.* Analysis of HDFS under HBase: A facebook messages case study. In *Proc. the 12th USENIX Conference on File and Storage Technologies*, February 2014, pp.199-212.

[6] Lepers B, Balmau O, Gupta K *et al.* KVell: The design and implementation of a fast persistent key-value store. In *Proc. the 27th ACM Symposium on Operating Systems Principles*, October 2019, pp.447-461. DOI: 10.1145/3341301.3359628.

[7] Wang Y, Tan J, Mao R *et al.* Temperature-aware persistent data management for LSM-Tree on 3-D NAND flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020, 39(12): 4611-4622. DOI: 10.1109/TCAD.2020.2982623.

[8] Lu L, Pillai S T, Arpaci-Dusseau C A *et al.* WiscKey: Separating keys from values in SSD conscious storage. *ACM Transactions on Storage*, 2017, 13(1): 1-28. DOI: 10.1145/3033273.

[9] Li Y, Chan H, Lee P *et al.* HashKV: Enabling efficient updates in KV storage via hashing. In *Proc. the 2018 USENIX Annual Technical Conference*, June 2018, pp.1007-1019. DOI: 10.5555/3277355.3277451.

[10] Raju P, Kadekodi R, Chidambaram V *et al.* PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proc. the 26th Symposium on Operating Systems Principle*, October 2017, pp.497-514. DOI: 10.1145/3132747.3132765.

[11] Chen S, Jin Q. Persistent B^+-trees in non-volatile main memory. *Proc. the VLDB Endowment*, 2015, 8(7): 786-797. DOI: 10.14778/2752939.2752947.

[12] Lee B, Ipek E, Mutlu O *et al.* Phase change memory architecture and the quest for scalability. *Communications of the ACM*, 2010, 53(7): 99-106. DOI: 10.1145/1785414.1785441.

[13] Zhou P, Zhao B, Yan J *et al.* A durable and energy efficient main memory using phase change memory technology. In *Proc. the 36th Annual International Symposium on Computer Architecture*, June 2009, pp.14-23. DOI: 10.1145/1555754.1555759.

[14] Yu S. Resistive Random Access Memory (RRAM). Morgan & Claypool, 2016. 10.2200/S00681ED1V01Y201510EET006.

[15] Apalkov D, Khvalkovskiy A, Watts S *et al.* Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems*, 2013, 9(2): Article No. 13. DOI: 10.1145/2463585.2463589.

[16] Venkataraman S, Tolia N, Ranganathan P *et al.* Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. the 9th USENIX Conference on File and Storage Technologies*, February 2011, pp.61-75.

[17] Yang J, Wei Q, Cheng C *et al.* NV-tree: Reducing consistency cost for NVM-based single level systems. In *Proc. the 13th USENIX Conference on File and Storage Technologies*, February 2015, pp.167-181. DOI: 10.5555/2750482.2750495.

[18] Oukid I, Lasperas J, Nica A *et al.* FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proc. the 2016 International Conference on Management of Data*, June 26–July 1, 2016, pp.371-386. DOI: 10.1145/2882903.2915251.

[19] Comer D. Ubiquitous B-tree. *ACM Comput. Surv.*, 1979, 11(2): 121-137. DOI: 10.1145/356770.356776.

[20] Bayer R. Binary B-trees for virtual memory. In *Proc. the 1971 ACM SIGFIDET Workshop on Data Description, Access and Control*, November 1971, pp.219-235. DOI: 10.1145/1734714.1734731.

[21] Ni J, Hu W, Li G *et al.* B^p-tree: A predictive B^+-tree for reducing writes on phase change memory. *IEEE Transactions on Knowledge and Data Engineering*, 2014, 26(10): 2368-2381. DOI: 10.1109/TKDE.2014.5.

[22] Hwang D, Kim W H, Won Y *et al.* Endurable transient inconsistency in byte-addressable persistent B+-tree. In *Proc. the 16th USENIX Conference on File and Storage Technologies*, February 2018, pp.187-200. DOI: 10.5555/3189759.3189777.

[23] Silberschatz A, Korth H, Sudarshan S. Database Systems Concepts (5th edition). McGraw-Hill, 2005.

[24] Dulloo R S, Kumar S, Keshavamurthy A *et al.* System software for persistent memory. In *Proc. the 9th European Conference on Computer Systems*, April 2014, Article No. 15. DOI: 10.1145/2592798.2592814.

[25] Volos1 H, Magalhaes G, Cherkasova L *et al.* Quartz: A lightweight performance emulator for persistent memory software. In *Proc. the 16th Annual Middleware Conference*, November 2015, pp.37-49. DOI: 10.1145/2814576.2814806.

[26] Cooper B F, Silberstein A, Tam E *et al.* Benchmarking cloud serving systems with YCSB. In *Proc. the 1st ACM Symposium on Cloud Computing*, June 2010, pp.143-154. DOI: 10.1145/1807128.1807152.

Rui-Xiang Ma is currently pursuing his Ph.D. degree with the Wuhan National Laboratory for Optoelectronics (WNLO), Huazhong University of Science and Technology (HUST), Wuhan. His current research interests include intelligent storage, machine learning, and non-volatile memory.

Fei Wu received her B.S. and M.S. degrees in electrical automation, control theory, and control engineering from Wuhan Industrial University, Wuhan, in 1997 and 2000, respectively, and her Ph.D. degree in computer science from Huazhong University of Science and Technology (HUST), Wuhan, in 2005, where she is currently a professor with the Information Storage Laboratory, Wuhan National Laboratory for Optoelectronics. Her research interests include computer architecture, non-volatile storage, and green storage. She is a senior member of CCF and a member of information storage of the China Computer Society.

Bu-Rong Dong received her B.S. degree in computer science from the Department of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, in 2020. She is pursuing her M.S. degree with Wuhan National Laboratory for Optoelectronics (WNLO), Huazhong University of Science and Technology, Wuhan. Her current research interests include nonvolatile memory technologies and index technology.

Meng Zhang received his Ph.D. degree in computer science from Wuhan National Laboratory for Optoelectronics (WNLO), Huazhong University of Science and Technology, Wuhan, in 2019. He is working as a postdoctoral researcher with WNLO, Huazhong University of Science and Technology, Wuhan. His current research interests include ECCs (Error Correction Code), applications of ECC in nonvolatile memory technologies, flash memory reliability, and NVM storage systems.

Wei-Jun Li received his B.S. degree in electronics engineering from the Beijing Institute of Technology, Beijing, in 1998, and his Ph.D. degree in microelectronics and solid state electronics from the Institute of Semiconductors, Chinese Academy of Sciences, Beijing, in 2004. He is currently a professor with the Laboratory of Artificial Neural Networks and High-Speed Circuits, Institute of Semiconductors, Chinese Academy of Sciences, Beijing. He is also the CTO of Shenzhen DAPU Microelectronics Co. Ltd.. He has authored or co-authored about 50 peer-reviewed journals and conference papers. His current research interests include machine learning, pattern recognition, and intelligent system.

Chang-Sheng Xie received his B.S. and M.S. degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, in 1982 and 1988, respectively, where he has served as the deputy director and is currently a professor with Wuhan National Laboratory for Optoelectronics, Wuhan. His research interests include new storage technology and architecture, multimedia computing and networks, computer storage, and network storage and security. He has served as the deputy director of the Computer Peripheral Equipment Committee of CCF, the committee member of information storage of the China Computer Society, and the vice chairman of the China Expert Committee of the International Network Storage Industry Association.