ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Master's Thesis

NVB-tree: Failure-Atomic B+-tree for Persistent Memory

Kibeom Jin

Department of Computer Science and Engineering

Graduate School of UNIST

2017

NVB-tree: Failure-Atomic B+-tree for Persistent Memory

Kibeom Jin

Department of Computer Science and Engineering

Graduate School of UNIST

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

NVB-tree: Failure-Atomic B+-tree for Persistent Memory

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Kibeom Jin

6. 13. 2017

Approved by

[illegible]

Advisor
Beomseok Nam

NVB-tree: Failure-Atomic B+-tree for Persistent Memory

Kibeom Jin

This certifies that the thesis of Kibeom Jin is approved.

6. 13. 2017

signature

[illegible]

Advisor: Beomseok Nam

signature

[illegible]

Young-Ri Choi: Thesis Committee Member #1

signature

[illegible]

Jiwon Seo: Thesis Committee Member #2

Abstract

Emerging non-volatile memory has opened new opportunities to re-design the entire system software stack and it is expected to break the boundaries between memory and storage devices to enable storage-less systems. Traditionally, B-tree has been used to organize data blocks in storage systems. However, B-tree is optimized for disk-based systems that read and write large blocks of data. When byte-addressable non-volatile memory replaces the block device storage systems, the byte-addressability of NVRAM makes it challenge to enforce the failure-atomicity of B-tree nodes.

In this work, we present NVB-tree that addresses this challenge, reducing cache line flush overhead and avoiding expensive logging methods. NVB-tree is a hybrid tree that combines the binary search tree and the B+-tree, i.e., keys in each NVB-tree node are stored as a binary search tree so that it can benefit from the byte-addressability of binary search trees. We also present a logging-less split/merge scheme that guarantees failure-atomicity with 8-byte memory writes. Our performance study shows that NVB-tree outperforms the state-of-the-art persistent index - wB+-tree by a large margin.

Contents

List of Figures

List of Algorithms

I. Introduction

Recent advances of byte-addressable non-volatile memories (NVRAM), such as 3D Xpoint [1], phase-change memory, and STT-MRAM are expected to open up new opportunities to transform computer main memory from volatile devices to persistent primary storage devices. NVRAM appears to provide low-latency reads and writes like DRAM, but it persists data in the unit of bytes and provides large storage capacity as in SSD [7, 9, 18, 5, 11]. In order to leverage high performance, persistence, and byte-addressability of NVRAM, various opportunities are being pursued in numerous domains including database logging [9], database buffer caching [23], and indexing structures [12].

Legacy disk-based B-tree and its variants have been originally developed for disk-based storage systems. However, as the memory size has been dramatically increased, database community has developed various in-memory indexing structures including T-tree [14]. These binary search trees were shown to outperform B-tree and its variants when a CPU cache size was small. However, as a CPU cache size grows in modern processors, cache-conscious variants of B-trees such as CSS-tree [20], CSB-tree [21], and FAST [8], have been shown to perform faster than legacy binary indexing structures.

Recently, there have been several research efforts that tried to redesign B-trees for NVRAM [15, 24, 26, 4]. B-trees is the most popular indexing structure that has been used for various storage systems and database systems. As NVRAM is expected to break down the wall between the main memory and the secondary storage systems, we need an efficient indexing structure that benefits from both the byte-addressability of DRAM and the durability and large capacity of block device storage systems.

One of the key challenges in designing B-tree for NVRAM is that it is not trivial to enforce failure-atomicity while leveraging the byte-addressability of NVRAM. In legacy DRAM+storage environment, reads and writes occur in block granularity and tree nodes are cached in volatile DRAM. Using the cached tree nodes, we can update the tree nodes and flush them only when everything is consistent.

On the other hand, if B-tree is stored in NVRAM, the granularity of writes is expected to be 8-bytes or a cache line at maximum using hardware transactional memory [12]. Such a small write granularity makes it difficult to guarantee consistency of a B-tree because every 8-byte write must transform a consistent B-tree node into another consistent B-tree node. B-tree often inserts a key in the middle of an array for sorting. Such an insertion modifies a large portion of a B-tree node. However, modern processors cache memory writes and arbitrarily reorder them to save the bandwidth. Suppose a system crashed while it sorts the keys in a B-tree node. Some portions of the array could have been flushed while other portions have not been flushed. Such a B-tree node will be in an inconsistent state. To update a B-tree node in a correct order to guarantee consistency, we need to call additional instructions such as clflush and mfence. As a result, legacy B-tree requires a large number of expensive clflush and mfence instructions [24].

In order to resolve this problem, several persistent B-tree variants including NV-tree [26] and wB+-tree [4] employ append-only update strategy. However, a problem of the append-only update strategy is that it fails to preserve the ordering of keys in B-tree nodes and it requires an expensive brute-force scanning to search a key. Therefore, wB+tree manages additional metadata in each tree node to separately manage the ordering of keys. Using the metadata, wB+-tree significantly reduced the number of cache line flushes. However, wB+-tree still calls at least four cache line flushes per insertion, which we believe is sub-optimal.

In this work, we design and implement NVB-tree, which is a variant of persistent B-tree that minimizes the number of memory write and cache line flush operations. To guarantee failure-atomicity and consistency with byte-addressable NVRAM, NVB-tree employs a binary search tree as an internal tree node format. By managing entries as a binary search tree, we can add, update, or delete entries with a failure-atomic 8-byte update operation. Another benefit of using a binary search tree is that it avoids brute-force scanning and improves search performance.

Besides the failure-atomic updates to a single tree node, NVB-tree eliminates the necessity of expensive logging for multiple tree node updates, i.e., a node split and a node merge. When a B-tree node overflows, the legacy B-tree split algorithm requires updates to at least three nodes - the node that overflows, its new sibling node, and their parent node. To guarantee atomicity and consistency of a tree structure, legacy split algorithms have used the expensive logging or journaling to keep track of changes. However, we propose to use a sibling pointer to make each memory write operation of a node split or a node merge process failure-atomic and consistent.

The contributions of this work are as follows.

- Non-volatile memory promises to encompass all the desirable features of memory and storage systems. For such non-volatile memory devices, we design and implement NVB-tree - a hybrid indexing structure that combines the benefits of binary search trees and B-trees, i.e., NVB-tree benefit from the failure-atomicity and byte-addressability of an in-memory index and the durability, cache line consciousness, and balanced tree height of a disk-based index.
- Second, we propose a logging-less failure-atomic split and merge algorithms for NVB-tree that eliminates the redundancy caused by logging or journaling. When multiple B+-tree nodes need to be modified, previous studies used redo/undo logging to provide failure atomicity. Our NVB-tree virtually combines multiple tree nodes and makes them behave as if they are a single node. Thereby, the failure-atomicity and consistency can be guaranteed by the 8-byte memory write operations without any additional mechanisms.
- Though extensive performance study, we show that our NVB-tree consistently outperforms wB+-tree by a large margin in terms of insertion time, search time, deletion time, cache line flushes, cache line accesses, and LLC misses. NVB-tree also shows a comparable performance with FPtree, which is designed for the hybrid NVRAM+DRAM memory environment.

Although NVB-tree is outperformed by FPtree in terms of insertion performance because NVB-tree stores all tree nodes in NVRAM, NVB-tree does not require recovery when a system crashes because any memory write operation in NVB-tree is failure-atomic.

The rest of this paper is organized as follows: In section 2, we present previous works that are most relevant to ours. In section 3, we present the structure and algorithms of our proposed NVB-tree. In section 4, we evaluate the performance of NVB-tree against state-of-the-art B+-tree variants for NVRAM. Finally, we conclude the paper in section 5.

II. RELATED WORK

Next generation byte-addressable non-volatile memory is now in the horizon. Anticipating its realization, numerous studies have been conducted on ways of exploiting its beneficial features [16, 17, 27]. In particular, its use has opened up new opportunities and challenges in redesigning file systems and database management systems by leveraging the durability and high performance of persistent memories [6, 9, 10, 11, 24, 26]. In this section, we review some previous work we deem most relevant to our study.

A few studies have proposed methods for safeguarding against failures for particular data structures and components of systems. Venkataraman et al. propose CDDS, a version-based recovery method for NVRAM [24]. Version based recovery methods, though, have a limitation in that it requires an expensive garbage collection for recovery. Yang et al. propose NV-Tree that makes updates to data structures in append-only manner so that it can roll back to its previous state without creating a journal [26]. FPTree, proposed by Oukid et al., is similar to NV-Tree in that it also stores leaf nodes in NVRAM while keeping internal tree nodes in DRAM [19]. Dierent from NV-Tree, FPTree exploits hardware transactional memory to efficiently handle concurrency of internal node accesses and it reduces the cache miss ratio via fingerprinting, which is one-byte hash for keys in each leaf node. Both NV-Tree and FPTree leverage both DRAM and NVRAM in their indexing structures. That is, they keep internal nodes in volatile memory while leaf nodes in NVRAM. Therefore, the internal nodes in NV-Tree and FP-Tree may be lost upon system failure. Although the internal nodes can be reconstructed from scratch using the leaf nodes in NVRAM, the entire reconstruction process can hinder the instant use of the index.

Also, when the index was being accessed by a persistent process, it can be a problem. Chen and Jin propose wB+-tree [4], an improved version of NV-Tree, that takes the same append-only approach but stores both leaf and internal nodes in NVRAM. Since B+-tree requires the entries in tree nodes to be sorted, updating the tree makes a large portion of the updated node dirty, which results in a large number of cache line flushes as in CDDS B-tree. Instead of keeping the entries in a sorted order, wB+-tree adds a level of indirection by having a bitmap and a slot-array that serves as an index to the actual entries written in an append-only manner. Although wB+-tree significantly reduced the number of cache line flushes, it still calls a large number of cache line flushes because of the additional metadata. For an instance, suppose we insert a new entry into wB+-tree that uses the bitmap and slot-array scheme. First, we need to call a cache line flush to invalidate the slot-array. Second, we need to call one or more cache line flushes to insert the new entry. If the entry is larger than a cache line size, we need multiple cache line flushes. After the inserted entry is flushed, wB+-tree calls multiple cache lines flushes to update the slot-array so that the slot index is in a sorted order. Finally, wB+-tree calls the last cache line flush to validate the slot-array. In order to reduce the number of cache line flushes, they developed an optimized scheme, slot-only wB+-tree, which requires only two cache line flushes - one for the record update and the other for bitmap update. Although the optimized slot-only wB+-tree reduces the number of cache line flushes down to two, which is optimal, the slot-only wB+-tree limits the number of entries in a tree node to be no more than eight. The small node degree makes the height of slot-only wB+-tree taller, and its performance is no better than bitmap wB+-tree. Moreover, wB+-tree performs logging for node splits that significantly increases the number of cache line flushes. As a result, the performance gain achieved from the optimal cache line flush is masked by the high split overhead. Also, the bitmap and slot-array add a level of indirection to key-pointer access, which hurts search performance as well.

III. FAILUREATOMIC NVBTREE

In this section, we present NVB-Tree, which combines the benefits of a byte-addressable binary search tree and block-based B-tree.

3.1 Design Rationale

The design of NVB-Tree has the following three goals in mind:

- Minimize the Number of Cache Line Flushes. Cache line flushes are expensive not only for memory writes but also for memory reads. In modern processors, memory read access latency is often hidden due to CPU caches. However, NVRAM demands cache line flushes in order to persist memory writes. Because clflush invalidates the flushed cache lines in CPU caches, reducing the number of cache line flushes not only benefit write transactions but also read transactions. To address this problem, Intel recently introduced new cache line flush instructions such as clflushopt and clwb that do not invalidate the flushed cache lines in CPU caches. However, the cache line flush instructions are still sensitive to the high write latency of NVRAM.

  Keeping entries in a tree node sorted requires shifting entries, making the consistency cost very high because of a large number of cache line flushes [26, 24]. To avoid such shift operations, NV-tree and wB+-tree proposed to keep entries unsorted and insert or delete entries in an append-only manner. In NVB-tree, we avoid shift operations by employing a binary search tree as an internal tree node format. A binary tree keeps entries sorted without shift operations, thereby reducing the number of cache line flushes.
- Fast Search Performance. In database applications, search operations occur more frequently than insertions. Furthermore, search performance also affects insertion performance because an insertion requires searching a leaf node where we insert a record. Since searches are more frequent operations than insertions, our primary design goal is not to sacrifice search performance. Compared to other B+-tree variants, such as wB+-tree [4], that do not sort leaf node entries, NVB-tree maintains entries in a binary search tree to speed up search performance. By leveraging the binary search tree, NVB-tree does not need additional metadata, which requires an additional memory access to read each key in a sorted order.
- Logging-less Node Split/Merge for Failure Atomicity. NVB-tree aims to guarantee failure atomicity for multiple node updates without expensive logging, which incurs a significant overhead of writing redundant copies. To avoid expensive logging, NVB-tree utilizes a sibling pointer in each B+-tree node to easily recover from tree modifications as in B-link tree [13].

With sibling pointers and 8-byte failure-atomic writes, NVB-tree achieves failure atomicity for node splits and node merges, i.e., NVB-tree is atomically transformed from one consistent state to another consistent state even if a system crashes.

3.2 Node Structure of NVB-tree

NVB-tree is a hybrid indexing structure that contains a binary search tree (BST) inside each B-tree node.

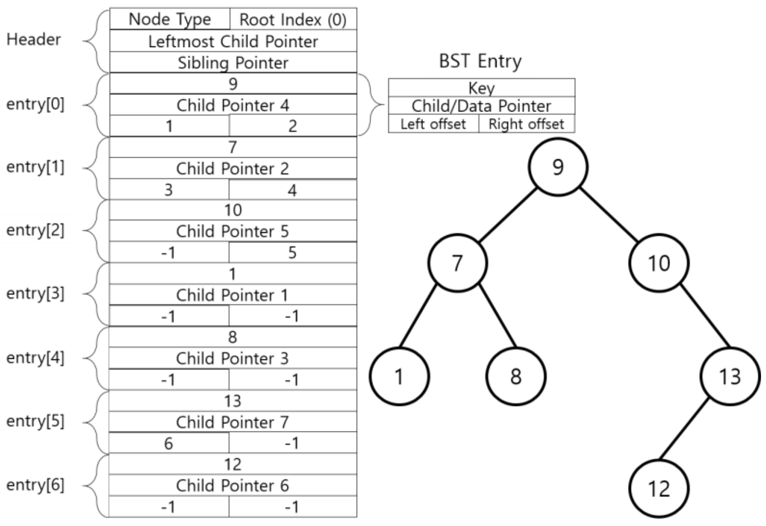


Figure 1: Node Structure of NVB-tree

Figure 1 illustrates the structure of an NVB-tree node. An NVB-tree node consists of a header and an array of BST entries. We refer to a node in a binary search tree as a BST entry to distinguish it from an NVB-tree node. Each BST entry has a key, a pointer to its child NVB-tree node, two offsets - one for the left child BST entry and the other for the right child BST entry. In the header, we have an offset field that points to the root BST entry, a field that indicates whether the node is an internal node or a leaf node, a sibling pointer, and a pointer to the leftmost child NVB-tree node. The pointer to the leftmost child NVB-tree node is used only if the node is a leaf node. In legacy B+-trees, an internal B-tree node does not have a sibling pointer. However, all NVB-tree node needs a sibling pointer to enable a virtual node, which is required to guarantee failure-atomicity. We will discuss the virtual node in section 3.4.4.

3.3 Failure-Atomic NVB-Tree Node Update

We now discuss how NVB-tree achieves failure-atomicity for a single NVB-Tree node update. For multiple node updates triggered by a node split or a node merge, we defer our discussion to Section 3.4.4 and Section 3.4.5.

3.3.1 Failure-Atomic Insertion into BST

An insertion into a binary search tree does not modify internal nodes but only adds a single leaf node. First, we search for a leaf node using the key of the new BST entry to be inserted. Next, we add the BST entry to the leaf node as the right or left child depending on the key.

Algorithm 1 Insert(key, ptr)

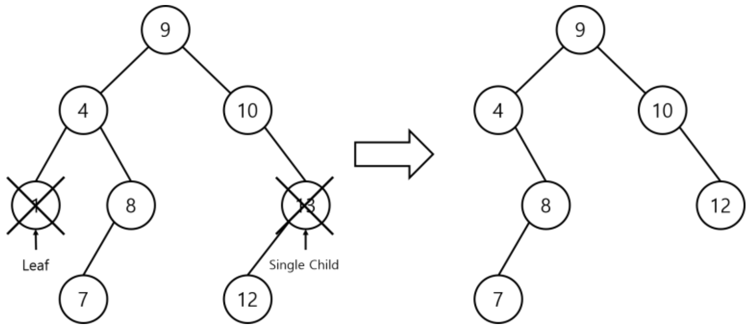
```
1: if count == MAX COUNT then
2:     count = lazyGarbageCollect(this);
3: if count == MAX COUNT then
4:     return SPLIT THIS NODE;
5: else
6:     idx = count;
7:     entry[idx].key = key;
8:     entry[idx].ptr = ptr;
9:     entry[idx].left = -1;
10:    entry[idx].right = -1;
11:    mfence();
12:    clflush(&entry[idx], sizeof(entry));
13:    count++;
14:    mfence();
15:    clflush(&count, sizeof(count));
16:    parent = searchBST(key);
17:    if key < entry[parent].key then
18:        entry[parent].left = idx;
19:        mfence();
20:        clflush(&entry[parent].left, 1);
21:    else
22:        entry[parent].right = idx;
23:        mfence();
24:        clflush(&entry[parent].right, 1);
```

Algorithm 1 shows the insertion algorithm of NVB-tree, which can be divided into three phases.

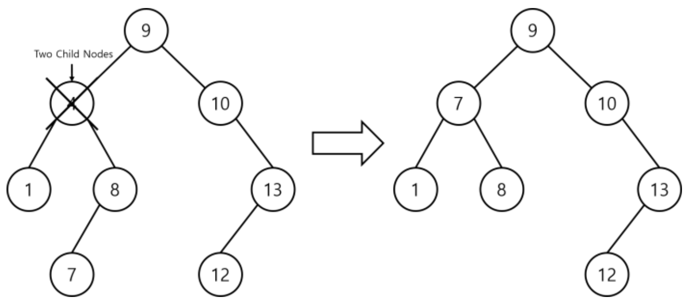
First, we store a new BST entry at the end of the array in an append-only manner (line 6-10). Next, we increase the size of the array for BST entries (line 13). Finally, we search the BST to find out the leaf node which will become the parent of the new BST entry (line 15). Depending on the key, we add the new BST entry as a left or right child of the parent entry (line 17-24). The ordering of these memory writes must be preserved for failure-atomicity. Therefore, we call memory fence and cache line flush instructions between each phase. Because byte-addressable NVRAM is not commercially available yet, we resort to discussing the recovery algorithm of NVB-tree, instead of physical power-off tests, under various failure cases that can occur.

First, suppose a system crashes while a new BST entry is being written (line 6-10). Although a system crashes in any step of this phase, it does not hurt consistency because the new entry is not added to the BST yet and the size of the array is not changed. If a system crashes after we increase the size of the array but before we add the BST entry to a leaf node of the BST, the BST entry will occupy an element of the array although the entry is not accessible from the BST. Such a memory leak problem does not lead to consistency problems but it will waste memory space. We resolve such a memory leak problem in a lazy manner. I.e., if a node overflows, we scan all the BST entries in the node to find out if there is any entry that is not accessible from the BST (line 2). If we find it, we reclaim the entry and decrease the size of the array (line 2) so that we can use it for a new BST entry. If there is no memory leak, we split the node (line 4). With such a lazy garbage collection scheme, we can reduce the overhead of maintaining tree nodes consistent while avoiding a full scan of NVB-tree nodes for recovery. Next, we add a new BST entry to the BST by a single failure-atomic 8-byte write operation. Any memory writes before line 16 will be ignored or garbage collected in our insertion algorithm. But, writing a child offset in the parent BST entry behaves as a commit mark of the insert transaction. If the offset is not updated, the new entry is not accessible. Hence, insertion into an NVB-tree node is failure-atomic as long as writing an offset is failure-atomic.

3.3.2 Failure-Atomic Deletion From BST



(a) Deleting a BST Node with No More than a Single Child



(b) Deleting a BST Node with Two BST Child Nodes

Figure 2: Failure-Atomic Deletion in NVB-tree Node

A deletion of a BST entry in a BST is more complicated than an insertion because the entry to be deleted may have child entries as shown in Figure 2. A deletion operation can be divided into two phases. In the first phase, we traverse the BST of an NVB-tree node to find an entry to be deleted (m) and track its parent entry (p). In the second phase, we check how many child entries m has.

- If m has no child (3 in Figure 2(a)), p's corresponding child offset is atomically updated to a sentinel value, such as -1 or NULL. Then, we flush it to NVRAM with mfence() and clflush(). This step requires a single cache line flush, which guarantees the failure-atomicity of NVB-tree node.
- If m has a single child c (13 in Figure 2(a)), the corresponding child offset of p is atomically set to the offset of c. Then, we flush it to NVRAM with mfence() and clflush(). This step also requires a single cache line flush, and it guarantees the failure-atomicity of NVB-tree node.
- If m has both left and right child entries (l and r), we allocate a new BST entry n and set its child offsets to l and r. In the sub-tree rooted at r, we find the leftmost entry (the smallest key in the right sub-tree) r2 and copy its key and value to n. We now flush n. Note that n is not added to the BST yet. Next, p's corresponding child offset is updated to point to n, which is then committed to NVRAM. Now we need to delete the smallest key in the right sub-tree.
    1. if r2 has no child, the child offset of r2's parent is set to a sentinel value, which is then committed to NVRAM.
    2. if r2 has a right child r3, the child offset of r2's parent is set to point r3, which is then committed to NVRAM.

The third case requires four cache line flushes because it uses copy-on-write. If a system crashes before p's child offset is set to n, we have a memory leak problem, but it can be fixed in a lazy manner when the node over flows, as was described in Section 3.3.1. If a system crashes after we update p's child offset but before we delete r2, we may end up with having duplicate BST entries in a BST. However, it should be noted that it does not hurt the correctness of a BST. Consider Figure 2(b), which shows a BST after we delete 4. If a system crashes before we delete 7, the BST may have 7 as a left child of 8 even though 8's parent is also 7. This redundancy wastes memory space but it does not violate the properties of BST and does not hurt the correctness of the BST. This is because the redundant entries in the BST will point to the same child node. We can also fix such a redundancy problem using the lazy garbage collection scheme. That is, only if we need a space in an NVB-tree node, we perform a full scan of BST and detect memory leaks and redundant entries to be deleted.

Algorithm 2 Delete(key)

```
1: del = searchBST(key, &parent);
2: if entry[del].left==-1 && entry[del].right==-1 then
3:     if entry[parent].left==del then
4:         entry[parent].left = =1;
5:         mfence();
6:         clflush(entry[parent].left, 1);
7:     else if entry[parent].right==del then
8:         entry[parent].right = =1;
9:         mfence();
10:        clflush(entry[parent].right, 1);
11:    else if entry[del].left==-1 || entry[del].right==-1 then
12:        if entry[del].left != -1 then
13:            child = entry[del].left;
14:        else
15:            child = entry[del].right;
16:        if entry[parent].left==del then
17:            entry[parent].left = child;
18:            mfence();
19:            clflush(entry[parent].left, 1);
20:        else
21:            entry[parent].right = child;
22:            mfence();
23:            clflush(entry[parent].right, 1);
24: else if entry[del].left!=-1 && entry[del].right!=-1 then
25:    if count == MAX COUNT then
26:        count = lazyGarbageCollect(this);
27:    else
28:        idx = count;
```

```
29:     minRight = getMinEntry(entry[del].right, &parentMinRight);
30:     entry[idx].key = entry[minRight].key;
31:     entry[idx].ptr = entry[minRight].ptr;
32:     entry[idx].left = entry[del].left;
33:     entry[idx].right = entry[del].right;
34:     mfence();
35:     clflush(&entry[idx], sizeof(entry));
36:     count++;
37:     mfence();
38:     clflush(&count, sizeof(count));
39:     **if** entry[parent].left == del **then**
40:         entry[parent].left = idx;
41:         mfence();
42:         clflush(&entry[parent].left, 1);
43:     **else**
44:         entry[parent].right = idx;
45:         mfence();
46:         clflush(&entry[parent].right, 1);
47:     **if** entry[minRight].right==-1 **then**
48:         entry[parentMinRight].left = -1;
49:     **else**
50:         entry[parentMinRight].left = entry[minRight].right;
51:     mfence();
52:     clflush(&entry[parentMinRight].left, 1);
```

The detail deletion algorithm is shown in Algorithm 2. If the entry to be deleted has only one or no child, recovery is not necessary. I.e., if a system fails before we flush the updated offset of the parent entry (line 2-23), the BST remains intact and it does not need any recovery. If a system fails while deleting an entry that has two child entries, it may incur a memory leak because we need an additional BST entry for the copy-on-write replacement. Suppose a system fails in between line 25-35. Since nothing has changed in the BST, it is in a consistent state. However, if a system crashes after we increase the count variable (line 39-46), we may have a memory leak if the replacement entry has not been added to the BST. Again, this memory leak will be detected and fixed by a lazy garbage collection scheme. Finally, if a system crashes while we delete the smallest entry in the right sub-tree (line 47-52), the BST may have a duplicate entry. But the duplicate entry in the right sub-tree can be ignored until it is detected and garbage collected by our lazy garbage collection scheme.

The deletion algorithm that we described so far leaves a hole in the BST entry array. Such holes degrade the node utilization and make a BST use more cache lines. Therefore, we better compact the array so that there are no holes in the middle. As we perform garbage collection, we create a new page and copy the elements from the original page to compact the array and reconstruct the balanced BST. Even if a system crashes during the procedure, it does not affect the correctness of the tree as the changes occur only in the new copy-on-write node. After the successful reconstruction, we flush the node and replace the original node. This can be done atomically by replacing the pointer in the Tree Node ID Mapping Table, which we describe in the Section 3.4.2. Note that this lazy garbage collection slightly hurts node insertion performance, but note that it is called only when we find a node has no available space for a new BST entry. Moreover, unless the write latency of NVRAM is many orders of magnitude slower than the read latency, improving the cache line utilization and reducing BST height help the insertion performance.

In summary, the deletion algorithm calls only one cache line flush instruction for its parent entry update when the deleted BST entry has one or no child entry. If the deleted entry has two child entries, it calls four cache line flush instructions due to the copy-on-write. If a BST is perfectly balanced, half of the entries will have two child entries and the rest of the entries will have no child. Ignoring the occasional overhead of our lazy garbage collection scheme, a deletion query will require at most 2.5 cache line flushes on average (n- = 2 x 1 + n=2 x 4). If a BST is not perfectly balanced, the average number of cache line flushes will be even smaller than 2.5.

3.4 Failure-Atomic Node Split and Merge

Insertions and deletions occasionally cause a split or a merge of tree nodes. If a split or a merge occurs, legacy B+-tree algorithms and even recently proposed persistent B+-trees such as wB+-tree rely on redo or undo logging. This is because a node split or a merge operation modifies multiple tree nodes. Even if we use the BST as the internal format of a tree node, it does not guarantee the failure-atomicity of a transaction that involves multiple node updates.

3.4.1 Sibling ID

However, if we provide an additional method for reaching each tree node and carefully split or merge B+-trees in a particular order, we can update multiple tree nodes in a failure-atomic manner. As in B-link tree [13], all NVB-tree nodes have a sibling ID that connects all nodes at a particular level as a linked list. When a node splits, a single node is replaced by two new nodes. When two nodes merge, a single node replaces the two nodes. That is, the sibling ID in each node allows to combine the current node and its right sibling node such that they become functionally the same as a single node until their parent node adds or deletes the child ID of the right sibling node. Note that we store child IDs instead of child pointers in tree nodes because we need a level of indirection to access any NVB-tree node for failure-atomicity. With the indirect access scheme, we can use copy-on-write to create a shadow copy and construct a balanced BST even when a tree node is referenced by multiple tree nodes.

3.4.2 Tree Node ID Mapping Table

problem of using sibling pointers and the copy-on-write in B+-tree is that the sibling pointer in the left sibling node and the child pointer in the parent node cannot be atomically updated. That is, if a node is replaced by a new node, its left sibling node and its parent node must atomically update their pointers, which is not possible. To resolve this problem, we store a tree node ID instead of a pointer in NVB-tree nodes, making use of an external tree node ID mapping table that stores the mapping between a tree node ID and its memory address. When the copy-on-write replaces a tree node, we do not update the pointers in its left sibling and its parent node, but we atomically overwrite the memory address in the table so that the left sibling and the parent node can access the updated tree node via its tree node ID and the mapping table.

3.4.3 Rebalancing BSTs

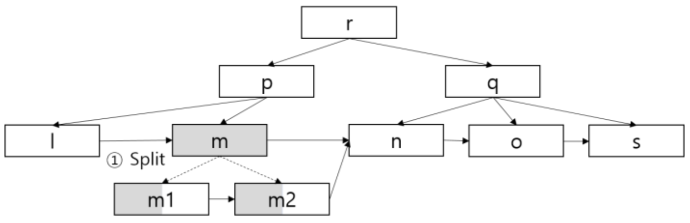
If an NVB-tree node split, a half of its entries need to be moved to a new sibling node. An NVB-tree node is a BST. Splitting a BST in the middle without restructuring often results in two unbalanced BSTs. It hurts not only search performance but also insertion performance as well. Besides this, splitting an NVB-tree node is likely to modify almost the entire tree node even if we perform an in-place update. Therefore, instead of the in-place update, we perform copy-on-write such that we construct two balanced BSTs by traversing partitioned BSTs in in-order and store the result in the BST entry array. This rebalancing algorithm takes O(n) time where n is the number of BST entries. However, we perform this expensive rebalancing operation only when an NVB-tree node split or two nodes merge so that the additional cache line flush overhead can be hidden because they need to be flushed anyway.

In addition to binary search trees, there exist various other byte-addressable in-memory indexing structures such as AVL-tree that keep the height of sub-trees balanced. However, AVL-tree performs rotation operations, which modify multiple BST entries and require multiple cache line flushes as are necessary for the deletion of a BST entry with two child entries.

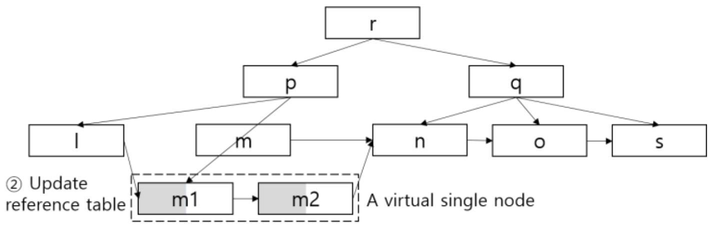
Although the depth of leaf nodes in a BST are not the same, the number of BST entries in an NVB-tree node is bounded by the size of an NVB-tree node. When the NVB-tree node size is 4 KB, maximum

254 8-byte keys can be indexed and the height of BST cannot be higher than 254. However, although the skewness of BSTs in NVB-trees is bounded, skewed BSTs degrade the tree traversal performance, and we can improve the search performance by rebalancing BSTs. Rebalancing BSTs causes a trade-off between search performance and insertion performance because of the expensive copy-on-write operation. However, tree traversal is the critical path and is much more frequent than insertions, we perform rebalancing at the cost of the copy-on-write.

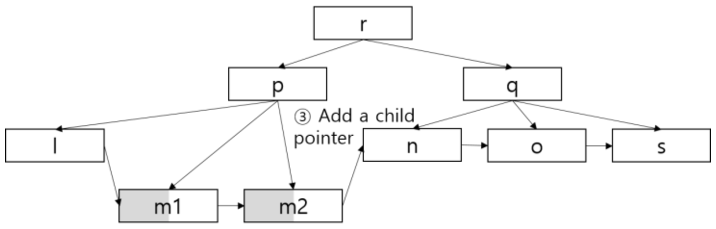
3.4.4 Node Split



(a) Copy-on-Write to Split an Overflown Node



(b) Update Parent's Child Pointer and the Left Node's Sibling Pointer



(c) Add a Right Split Node in the Parent Node

Figure 3: Failure-Atomic Node Split in NVB-tree Node

Figure 3 illustrates the node split algorithm of NVB-tree. To guarantee consistency, we do not consider the entry that caused the overflow during the split process. Once we finish the node split operation and the NVB-tree is consistent, we insert the new entry into one of the split leaf nodes.

First, i) we allocate two new nodes (m1 and m2) and construct a balanced binary search tree in each of them with partitioned keys. Having split the node m, ii) we replace the memory address of m with the memory address of m1 in the tree node ID mapping table as shown in Figure 3(b). By doing so, we can make both the left sibling node (l) and its parent node (p) point to m1 in a failure-atomic way.

A drawback of this approach is that we need to look up the tree node ID mapping table every time we need to access a tree node. Such an indirect access can cause some performance overhead and management overhead. But the number of accesses to the tree node ID mapping table per query is limited by the height of NVB-tree, which is very small.

Suppose a system crashes after we replace the memory address of m with the memory address of m1 in the tree node ID mapping table as shown in Figure 3(b). If a query is looking for an entry in m2, the query will access p first and then m instead of m2 because m2 is not added to p yet.

At the first glance, this status seems inconsistent and incorrect. But we can easily resolve this problem by making a search algorithm visit the right sibling node if a given search key is greater than the largest key in the current node. That is, a sibling pointer can make multiple sibling nodes virtually a single node. In the example, m and m2 are a virtual single node.

Next, iii) we add a new right sibling node m2 to the parent node (p), which completes the node split process. Finally, iv) we insert the entry that caused the overflow.

Algorithm 3 SplitNode(p, ent, n)

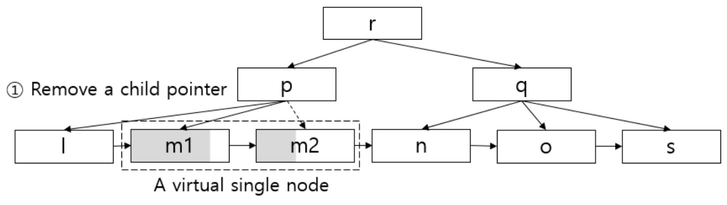
```
1: n1 = alloc();
2: n2 = alloc();
3: m = findMedian(n);
4: copyEntries(n, 0, m, n1);
5: copyEntries(n, m, 1, n2);
6: n1.sibling = &n2.id;
7: n2.sibling = n.sibling;
8: mfence();
9: clflush(&n1, sizeof(page))
10: clflush(&n2, sizeof(page))
11: nodeIdTable[n.id] = &n1;
12: mfence();
13: clflush(&nodeIdTable[n.id], 8);
14: p.Insert(m, &n2);
```

Algorithm 3 shows the pseudo code that describes the details of the NVB-tree split algorithm.

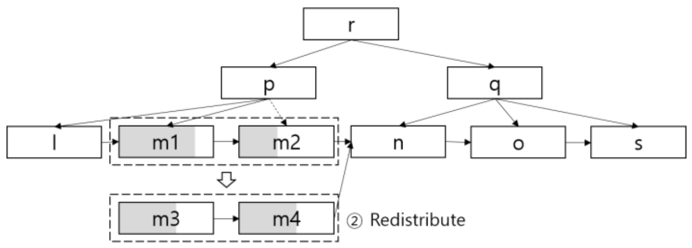
3.4.5 Node Merge and Redistribution

If the node utilization of a tree node becomes lower than a threshold value (e.g., 50% in our implementation), we check if the entries in the node and a sibling fit into a single node. If so, we merge the two nodes. Otherwise, we migrate some entries from the sibling node to the under-utilized node.

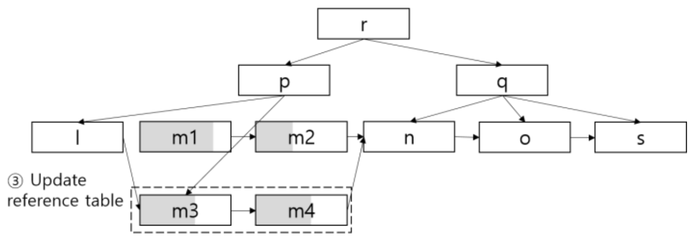
A node merge or a redistribution is similar to a node split, but the order of operations is reversed.



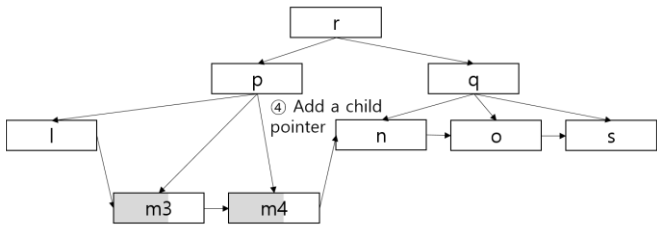
(a) Copy-on-Write to Split an Overflow Node



(b) Update Parent's Child Pointer and the Left Node's Sibling Pointer



(c) Add a Right Split Node in the Parent Node



(d) Replace Parent's Child Pointer with the Left Split Node

Figure 4: Failure-Atomic Redistribution in NVB-tree Node

Figure 4 shows how we redistribute the entries from under-utilized nodes to balance the number of entries. A node merge also follows the same steps except that it creates a single node instead of two.

First, i) we delete the pointer to the right sibling node to be merged or redistributed (m2 in Figure 3(b)) from its parent node (p). This step will again make m1 and m2 a virtual single node. Next, ii) we allocate a new node (or two nodes) and copy the entries from the two under-utilized nodes. Again, in this step, we sort the entries and construct a balanced BST (or two balanced BSTs) in the new node(s). Then, iii) we atomically update the tree node ID mapping table with the memory address of the merged (or redistributed) tree node so that the left sibling and the parent node point to the updated tree node. Finally, iv) we add the right sibling node to the parent node to complete the merge (or redistribution). The discussion of failure-atomicity and recovery is similar to that of node splitting and is omitted here.

IV. EVALUATION

4.1 Experimental Setup

We run the experiments on a workstation that has four Intel Haswell-EX E7-4809 v3 processors (2.0GHz, 8x32KB instruction cache, 8x32 KB data cache, 8x256 KB L2 cache, and 20 MB L3 cache) and 64 GB of DDR3 DRAM.

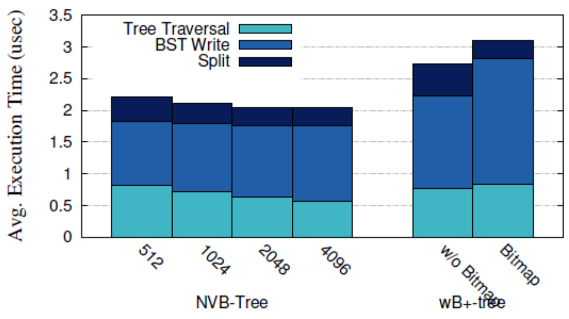
Byte-addressable non-volatile main memory is not commercially available yet. Thus, we emulate NVRAM using Quartz [25, 2], a DRAM-based NVRAM latency emulator [25]. Quartz consists of a kernel module and a user-mode library that models application-perceived latency by injecting stall cycles into each predefined time interval, called epoch. We configure minimum and maximum epochs to 5 nsec and 10 nsec for all our experiments. We adjust read latency of NVRAM from 300 nsec, which is minimum NVRAM latency that we can configure in our testbed environment. Quartz does not support write latency emulation in its public distribution. Therefore, we inject stall cycles after each clflush instructions to emulate write latency of NVRAM, as was done by [7, 9, 23, 12]. Note that write latency of NVRAM is often hidden by CPU cache. Hence, we do not add the software delay to store instructions. As for the NVRAM bandwidth, we assume that NVRAM bandwidth is the same with that of DRAM since Quartz does not allows us to emulate both latency and bandwidth at the same time.

We compare the performance of NVB-tree against wB+-tree and FPTree. Unlike NVB-tree and wB+-tree, FP-tree stores internal tree nodes in DRAM and leaf nodes in NVRAM. So FPTree is less sensitive to a NVRAM latency. However, if a system crashes and internal nodes are deleted from DRAM, it must reconstruct the entire tree structure from scratch using persistent leaf nodes. On the other hand, NVB-tree and wB+-tree store all internal nodes and leaf nodes in NVRAM. Hence, their failure-atomic node update algorithms eliminate the necessity of recovery process and thereby make instant recovery possible at the cost of slightly slow access to internal nodes. However, there exist many applications that can tolerate such a performance penalty as we will discuss in Section 4.4.2, and the difference between DRAM and NVRAM latency is expected to be less than an order of magnitude.

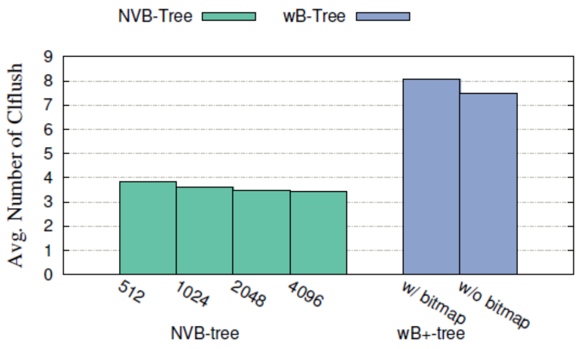
We implemented three variants of NVB-tree. The first one is the optimized NVB-tree with both array compaction and BST rebalancing optimizations (NVB-Tree). The second one is the NVB-tree with BST rebalancing but without the array compaction optimization (NVB(-c)). And, the last one does not use either of the two optimizations (denoted as NVB(-c,-r)).

The performance of binary search tree is affected by the ordering of inserted keys. If we insert keys in a sorted order, binary search trees will be skewed and the tree height will be the number of entries (O(n)), which is the worst case for binary search trees. Hence, we evaluate the performance of NVB-trees with two types of workloads - sequential key insertions and random key insertions. Unless specified, we use the random key workload.

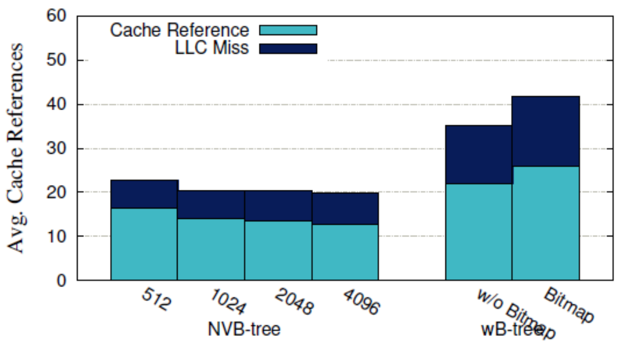
4.2 Node Size



(a) Insertion Time per Query



(b) Number of clflush per Query



(c) Number of Cache Line Accesses per Query

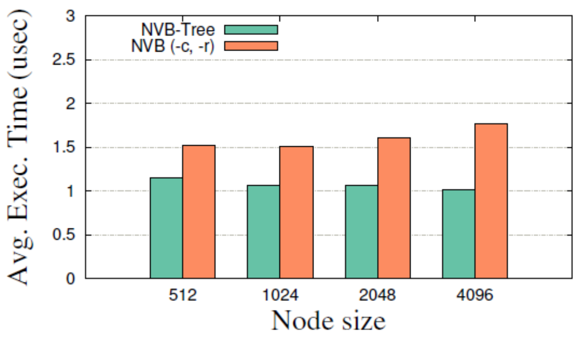
Figure 5: Insertion Performance with Varying Node Sizes

In the experiment shown in Figure 5, we insert 10 million key and value pairs in a random order while we vary the node size of each tree. We assume the latency of NVRAM is the same with that of DRAM in this experiment. We discuss the effect of NVRAM latency in Section 4.4.

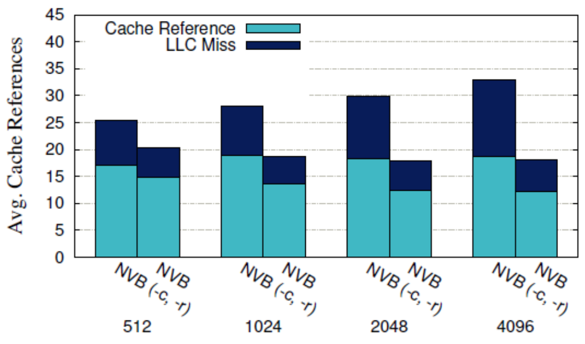
Figure 5(a) breaks down the time spent on the insertion operation. As we increase the node size, the insertion performance of NVB-tree slightly improves because of the shorter tree height and the reduced number of splits. On the other hand, the node size in wB+-tree is fixed. I.e., wB+-tree with bitmap can have up to 63 keys (1 KB node size) and wB+-tree without bitmap can have only 8 keys. Overall, NVB-tree is about 35% and 52% faster than wB+-tree without bitmap and wB+-tree with bitmap respectively in terms of the insertion time.

Figure 5(b) shows the average number of clflush instructions for the same experiments. While wB+-tree with bitmap calls about eight clflush instructions, NVB-tree calls less than four clflush instructions. wB+-tree calls more cache line flushes than NVB-tree because of the metadata update and also because it performs logging when splitting a node. Note that we can make NVB-tree call even a less number of clflush instructions if we do not use copy-on-write. But, as can be seen in Figure 5(a), improving tree traversal time is as important as improving tree node update time.

Figure 5(c) shows the number of CPU cache references and the number of LLC misses per query. NVB-tree accesses 43~53% smaller number of cache lines than wB+-trees. This is because NVB-tree does not allow holes in BST entry arrays, and thereby it increases the utilization of cache lines and reduces the number of cache lines. Besides this, NVB-tree dramatically decrease the LLC miss rate, mainly because of the reduces number of cache line accesses and a fewer number of clflush instructions that invalidate cache lines from LLC.



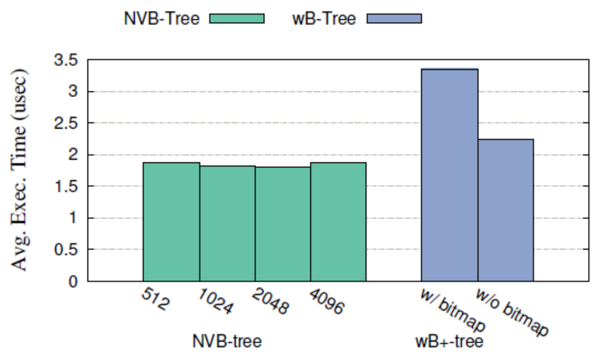
(a) Search Time per Quer



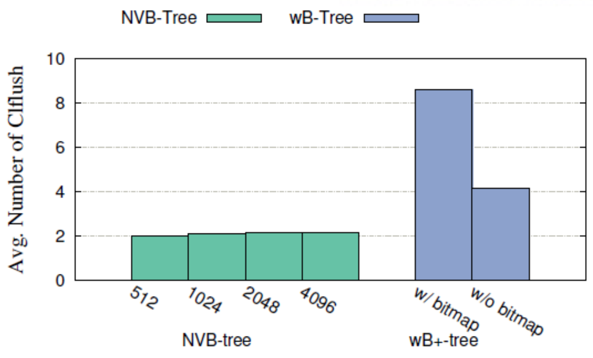
(b) Number of Cache Line Accesses per Query

Figure 6: Search Performance with Varying Node Sizes

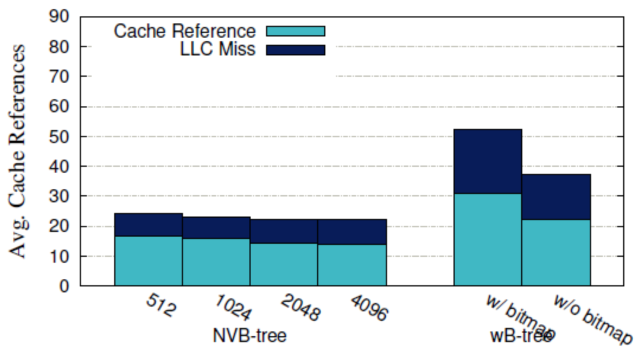
Figure 6 shows the search performance of NVB-trees with and without rebalancing optimization. As we increase the node size, the unbalanced BSTs degrades the search performance. But the rebalancing optimization helps mitigate the problem and the search performance even improves with larger tree node sizes because of shorter tree depths. Figure 6(b) shows the rebalancing optimization effectively reduces the number of accessed cache lines and the number of LLC misses.



(a) Deletion Time per Query



(b) Number of Cache Line Flushes



(c) Number of Cache Line Accesses per Query

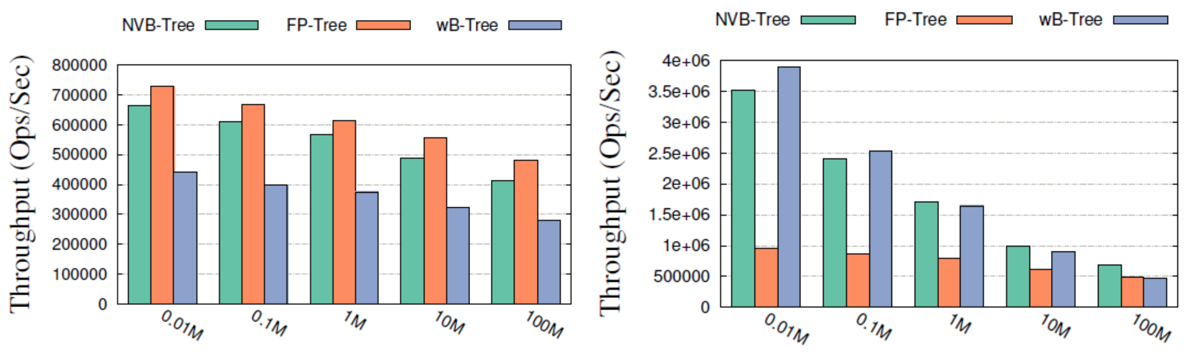
Figure 7: Deletion Performance with Varying Node Sizes

Figure 7 shows the deletion performance of NVB-tree. As in insertion and search performance, the deletion time of NVB-tree is almost independent of the node size. This is because the rebalancing optimization effectively balances BSTs so that a tree traversal takes O(n) time and the deletion of a BST entry takes a constant time. Without the rebalancing optimization, the insertion/search/deletion performance of NVB-tree degrades as we increase the node size.

Figure 7(b) shows that NVB-tree calls about 2 cache line flushes on average. Although the deletion algorithm of NVB-tree requires multiple 4 cache line flushes if the BST entry to be deleted has two child entries, it requires only a single cache line flush most of the time, i.e., when the entry has no or one child entry. Therefore, the average number of cache line flushes is much smaller than that of wB+-tree. Note that the lazy garbage collection reclaims all used BST entries only when a node overflows, which effectively hides the overhead of garbage collection.

The number of LLC misses with NVB-tree is also much smaller than that of wB+-tree as shown in Figure 7(c). This is because NVB-tree removes holes in the BST entry array and it calls a fewer number of clflush instructions that invalidate cache lines from LLC.

4.3 Throughput

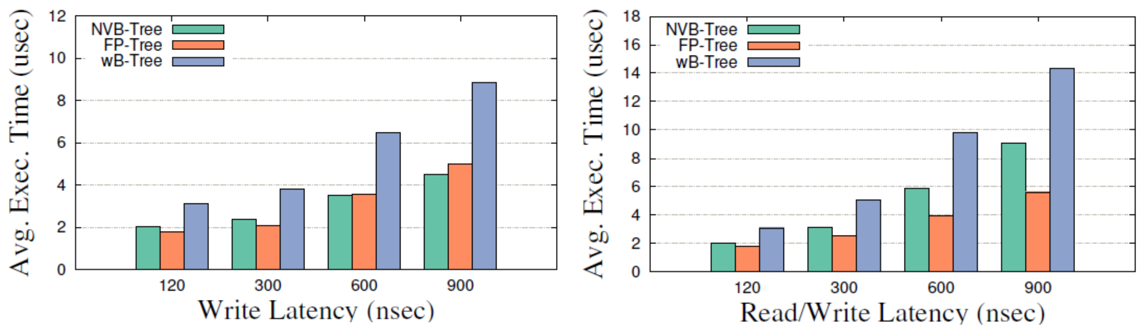


(a) Insertion Throughput (b) Search Throughput

Figure 8: Throughput with Varying Number of Indexed Keys

In the experiments shown in Figure 8, we create indexes with various number of key value pairs, and we measure the performance of NVB-tree against FPTree and wB+-tree. As for the insertion throughput, wB+-tree shows the lowest throughput while FP-tree yields slightly higher throughput than NVB-tree because FPTree does not have clflush overhead when updating internal nodes. However, when the number of indexed data is small, the search throughput of FPTree is 3.7x lower than that of NVB-tree. This is because most of the performance gain in FPTree comes from faster access to internal nodes in DRAM. I.e., FPTree keeps keys in internal nodes sorted, but keys in leaf nodes are unsorted. However, when the tree height is small, we access a fewer number of internal tree nodes and FPTree suffers from slow access to leaf nodes.

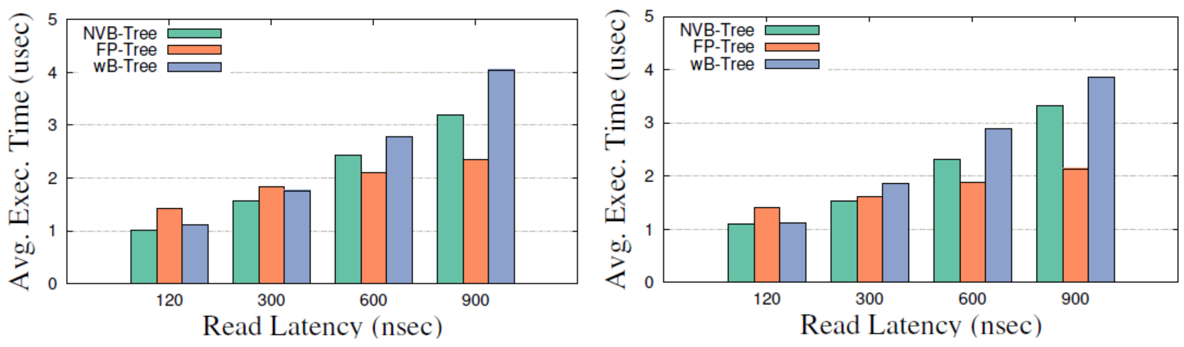
4.4 NVRAM Latency Effect



(a) Insertion with Varying Write Latency (b) Insertion with Varying R/W Latency

Figure 9: Insertion Performance Comparison (Latency)

Figure 9 shows the insertion performance of NVB-tree against wB+-tree and FPTree with varying the latency of NVRAM. If the read latency of NVRAM is the same with that of DRAM, NVB-tree and FPTree show comparable insertion performances and they consistently outperform wB+-tree. As the write latency increases, NVB-tree starts to show better performance than FPTree because NVB-tree calls a fewer number of cache line flushes than FPTree. However, if we increase both read and write latency, FPTree benefits from the lower latency of DRAM. As a result, FPTree consistently outperforms NVB-tree and the performance gap between them increases with a higher read latency.



(a) Randomly Inserted Keys (b) Sequentially Inserted Keys

Figure 10: Search Performance of Balanced BSTs vs Skewed BSTs

For the experiments shown in Figure 10, we measure the search performance of NVB-tree with two types of workloads; (a) we insert random 10 million key value pairs, which is the best case, and (a) we also create an NVB-tree with monotonically increasing 10 million key value pairs, which is the worst case as it will skew BSTs.

Even if we make a BST in a leaf node completely skewed by inserting sequential keys, splitting a node will rebalance the BSTs. Thus, the key distribution does not affect the tree height of BSTs in NVB-tree. Therefore, the performance of NVB-tree with two different workloads do not make a significant difference as shown in Figure 10. However, such a sequential insertion pattern can hurt the node utilization because the left split node will never be updated, which explains NVB-tree with sequential insertions performs slightly slower than random insertions.

When the read latency is lower than 600 nsec, NVB-tree shows the fastest search performance because NVB-tree performs the binary search that helps reduce the number of accessed cache lines and LLC misses. However, if read latency of NVRAM is higher than 600 nsec, FPTree benefit from fast access to internal nodes and it starts to outperform NVB-tree and wB+-tree.

4.4.1 Mixed Workload

Unless the read latency of NVRAM is 6x higher than that of DRAM, NVB-tree is faster than FPTree and wB+-tree in terms of search performance. However, in terms of insertion and deletion performance, NVB-tree is slower than FPTree although it is faster than wB+-tree.

In real-world applications, workloads are typically mixed with inserts, deletions, and searches. If the search is a more frequent operation and the insert and delete are less frequent operations, the slow insertion/deletion performance of NVB-tree can be masked by the fast search performance.

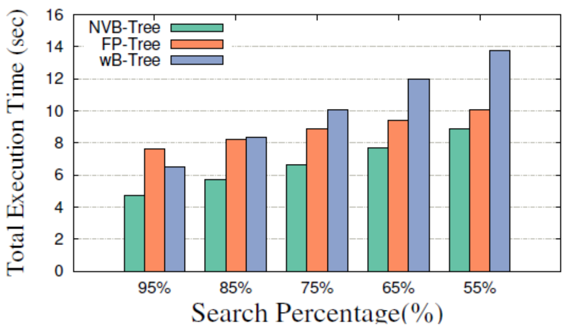


Figure 11: Performance Comparison (Mixed Workload)

In the experiments shown in Figure 11, we measure the query processing performance with mixed workloads while we vary the search and insertion ratio. We set the read and write latency of NVRAM to 300 nsec in these experiments. As we increase the insertion ratio, the batch query execution time increases as an insertion query takes much longer than a search. In particular, the performance gap between NVB-tree and FP-tree is decreased as we perform more insertions.

4.4.2 Discussions: NVBtree vs. FPTree

In summary, NVB-tree consistently outperforms wB+-tree - the state-of-the-art B+-tree variant for NVRAM by a large margin. Besides this, NVB-tree shows a comparable performance with FPTree - a B+-tree variant for hybrid DRAM+NVRAM environment.

Although FPTree is often faster than NVB-tree as we showed in our experiments, there exist some applications, such as BTRFS [22], that use B-trees as their main on-disk data structure and they cannot tolerate the volatility of internal tree nodes and the expensive tree reconstruction for system failures.

Another important use cases of NVRAM-only B+-tree structures exist in the memory-driven computing [3], where NVRAM is used as a shared memory pool while each computing node has some local memory. The capacity of such a shared NVRAM pool can be very large as NVRAM promises considerably high capacity at low cost. In such a memory-driven computing environment, we cannot store internal tree nodes in DRAM as it is not clear which computing node should have it in its local DRAM.

V. CONCLUSION

In this work, we presented NVB-tree - a variant of B+-tree for NVRAM that provides failure-atomicity without expensive logging. NVB-tree is a hybrid tree that combines the binary search tree and the B+-tree.

B+-tree requires keys to be sorted. However, the sorting makes B+-tree perform poor because it requires to shift a large number of keys in a tree node and it calls expensive cache line flush and memory fence instructions. In order to resolve the problem, NVB-tree organizes keys in a BST so that it can avoid shift operations. I.e., BST allows NVB-tree to take advantage of byte-addressability and to enforce failure-atomicity with the 8-byte memory write operations. NVB-tree also maintains the benefit of block-based B+-tree, i.e., balanced tree height, cache-awareness, and durability.

Another feature of NVB-tree that we discuss in this paper is the logging-less multiple node updates in NVRAM. Using the sibling pointer, we can detach/attach any tree node from/to its parent via 8-byte memory writes without hurting the correctness of B+-tree as in B-link tree. Leveraging this property, we carefully and atomically split or merge tree nodes with the help of an external tree node ID mapping table.

Our performance study shows that NVB-tree outperforms the state-of-the-art persistent index - wB+-tree by a large margin, and it also shows a comparable performance with DRAM+NVRAM-based FPTree that does not enable instant recovery.

REFERENCES

[1] Intel and Micron produce breakthrough memory technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology.

[2] Quartz. https://github.com/HewlettPackard/quartz.

[3] Hewlett Packard Enterprise - The Machine. https://www.labs.hpe.com/the-machine.

[4] Shimin Chen and Qin Jin. Persistent B+-Trees in non-volatile main memory. Proceedings of the VLDB Endowment (PVLDB), 8(7):786-797, 2015.

[5] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP), 2009.

[6] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. High performance database logging using storage class memory. In Proceedings of the 27th International Conference on Data Engineering (ICDE), pages 1221-1231, 2011.

[7] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. Nvram-aware logging in transaction systems. Proceedings of the VLDB Endowment, 8(4), 2014.

[8] Changkyu Kim and Jatin Chhugani and Nadathur Satish and Eric Sedlar and Anthony D. Nguyen and Tim Kaldewey and Victor W. Lee and Scott A. Brandt and Pradeep Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In Proceedings of 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD), 2010.

[9] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: Exploiting nvram in write-ahead logging. In 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2016.

[10] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 399-411, 2016.

[11] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST), 2013.

[12] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In Proceedings of the 15th USENIX conference on File and Storage Technologies (FAST), 2017.

[13] Philip L. Lehman and S. Bing Yao. Efficient Locking for Concurrent Operations on B-Trees. ACM Transactions on Database Systems, 6(4):650-670, December 1981.

[14] Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In Proceedings of the 12th International Conference on Very Large Data Bases (VLDB), 1986.

[15] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting hardware transactional memory in main-memory databases. In Proceedings of the 30th International Conference on Data Engineering (ICDE), 2014.

[16] Sparsh Mittal and Jeffrey S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. IEEE Transactions on Parallel and Distributed Systems, 27(5):1537-1550, June 2015.

[17] Sparsh Mittal, Jeffrey S. Vetter, and Dong Li. A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches. IEEE Transactions on Parallel and Distributed Systems, 26(6):1524-1537, June 2015.

[18] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In Proceedings of the 11th European Conference on Computer Systems (EuroSys 16), 2016.

[19] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In Proceedings of 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD), 2016.

[20] Jun Rao and Keneeth A. Ross. Cache conscious indexing for decision-support in main memory. In Proceedings of the 25th International Conference on Very Large Data Bases (VLDB), 1999.

[21] Jun Rao and Keneeth A. Ross. Making B+-Trees Cache Conscious in Main Memory. In Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD), 2000.

[22] Ohad Rodeh and Josef Bacik and Chris Mason. BTRFS: The Linux B-Tree Filesystem. ACM Transactions on Storage (TOS), 9(3), Article No. 9, August 2013. Efficient Locking for Concurrent Operations on B-Trees. ACM Transactions on Database Systems, 6(4):650-670, December 1981.

[23] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, Sam H. Noh. Failure-Atomic Slotted Paging for Persistent Memory. In 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017.

[24] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In 9th USENIX conference on File and Storage Technologies (FAST), 2011.

[25] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In 15th Annual Middleware Conference (Middleware '15), 2015.

[26] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, and Khai Leong Yong. NV-Tree: reducing consistency const for NVM-based single level systems. In Proceedings of the 13th USENIX conference

on File and Storage Technologies (FAST), 2015.

[27] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In Proceedings of the 31st International Conference on Massive Storage Systems (MSST), 2015.