WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems

Se Kwon Lee, *UNIST (Ulsan National Institute of Science and Technology);*
K. Hyun Lim, *Hongik University;* Hyunsub Song, Beomseok Nam, and Sam H. Noh,
UNIST (Ulsan National Institute of Science and Technology)

https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee

This paper is included in the Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

Open access to the Proceedings of the 15th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems

Se Kwon Lee, K. Hyun Lim[†], Hyunsub Song, Beomseok Nam, Sam H. Noh
UNIST (Ulsan National Institute of Science and Technology), [†]Hongik University

Abstract

Recent interest in persistent memory (PM) has stirred development of index structures that are efficient in PM. Recent such developments have all focused on variations of the B-tree. In this paper, we show that the radix tree, which is another less popular indexing structure, can be more appropriate as an efficient PM indexing structure. This is because the radix tree structure is determined by the prefix of the inserted keys and also does not require tree rebalancing operations and node granularity updates. However, the radix tree as-is cannot be used in PM. As another contribution, we present three radix tree variants, namely, WORT (Write Optimal Radix Tree), WOART (Write Optimal Adaptive Radix Tree), and ART+CoW. Of these, the first two are optimal for PM in the sense that they only use one 8-byte failure-atomic write per update to guarantee the consistency of the structure and do not require any duplicate copies for logging or CoW. Extensive performance studies show that our proposed radix tree variants perform considerable better than recently proposed B-tree variants for PM such NVTree, wB+Tree, and FPTree for synthetic workloads as well as in implementations within Memcached.

1 Introduction

Previous studies on indexing structures for persistent memory (PM) have concentrated on B-tree variants. In this paper, we advocate that the radix tree can be better suited for PM indexing than B-tree variants. We present radix tree variant indexing structures that are optimal for PM in that consistency is always guaranteed by a single 8-byte failure-atomic write without any additional copies for logging or CoW.

Emerging persistent memory technologies such as phase-change memory, spin-transfer torque MRAM, and 3D Xpoint are expected to radically change the landscape of various memory and storage systems [4, 5, 7, 9, 10, 14]. In the traditional block-based storage device, the failure atomicity unit, which is the update unit where consistent state is guaranteed upon any system failure, has been the disk block size. However, as persistent memory, which is byte-addressable and non-volatile, will be accessible though the memory bus rather than via the PCI interface, the failure atomicity unit for persistent memory is generally expected to be 8 bytes or no larger than a cache line [5, 6, 12, 13, 15, 19].

The smaller failure atomicity unit, however, appears to be a double-edged sword in the sense that though this allows for reduction of data written to persistent store, it can lead to high overhead to enforce consistency. This is because in modern processors, memory write operations are often arbitrarily reordered in cache line granularity and to enforce the ordering of memory write operations, we need to employ memory fence and cache line flush instructions [21]. These instructions have been pointed out as a major cause of performance degradation [3, 9, 15, 20]. Furthermore, if data to be written is larger than the failure-atomic write unit, then expensive mechanisms such as logging or copy-on-write (CoW) must be employed to maintain consistency.

Recently, several persistent B-tree based indexing structures such as NVTree [20], wB+Tree [3], and FPTree [15] have been proposed. These structures focus on reducing the number of calls to the expensive memory fence and cache line flush instructions by employing an append-only update strategy. Such a strategy has been shown to significantly reduce duplicate copies needed for schemes such as logging resulting in improved performance. However, this strategy does not allow these structures to retain one of the key features of B-trees, that is, having the keys sorted in the nodes. Moreover, this strategy is insufficient in handling node overflows as node splits involve multiple node changes, making logging necessary.

While B-tree based structures have been popular in-memory index structures, there is another such structure,

namely, the radix tree, that has been less so. The first contribution of this paper is showing the appropriateness and the limitation of the radix tree for PM storage. That is, since the radix tree structure is determined by the prefix of the inserted keys, the radix tree does not require key comparisons. Furthermore, tree rebalancing operations and updates in node granularity units are also not necessary. Instead, insertion or deletion of a key results in a single 8-byte update operation, which is perfect for PM. However, the original radix tree is known to poorly utilize memory and cache space. In order to overcome this limitation, the radix tree employs a path compression optimization, which combines multiple tree nodes that form a unique search path into a single node. Although path compression significantly improves the performance of the radix tree, it involves node split and merge operations, which is detrimental for PM.

The limitation of the radix tree leads us to the second contribution of this paper. That is, we present three radix tree variants for PM. For the first of these structures, which we refer to as Write Optimal Radix Tree for PM (WORTPM, or simply WORT), we develop a failure-atomic path compression scheme for the radix tree such that it can guarantee failure atomicity with the same memory saving effect as the existing path compression scheme. For the node split and merge operations in WORT, we add memory barriers and persist operations such that the number of writes, memory fence, and cache line flush instructions in enforcing failure atomicity is minimized. WORT is optimal for PM, as is the second variant that we propose, in the sense that they require only one 8-byte failure-atomic write per update to guarantee the consistency of the structure without any duplicate copies.

The second and third structures that we propose are both based on the Adaptive Radix Tree (ART) that was proposed by Leis et al. [11]. ART resolves the trade-off between search performance and node utilization by employing an adaptive node type conversion scheme that dynamically changes the size of a tree node based on node utilization. This requires additional metadata and more memory operations than the traditional radix trees, but has been shown to still outperform other cache conscious in-memory indexing structures. However, ART in its present form does not guarantee failure atomicity. For the second radix tree variant, we present Write Optimal Adaptive Radix Tree (WOART), which is a PM extension of ART. WOART redesigns the adaptive node types of ART and carefully supplements memory barriers and cache line flush instructions to prevent processors from reordering memory writes and violating failure atomicity. Finally, as the third variant, we present ART+CoW, which is another extension of ART that makes use of CoW to maintain consistency. Unlike B-tree variants where CoW can be expensive, with the radix tree, we show that CoW incurs considerably less overhead.

Through an extensive performance study using synthetic workloads, we show that for insertion and search, the radix tree variants that we propose perform better than recent B-tree based persistent indexes such as the NVTree, wB+Tree, and FPTree [3, 15, 20]. We also implement the indexing structure within Memcached and show that similarly to the synthetic workloads, our proposed radix tree variants perform substantially better than the B-tree variants. However, performance evaluations show that our proposed index structures are less effective for range queries compared to the B-tree variants.

The rest of the paper is organized as follows. In Section 2, we present the background on consistency issues with PM and PM targeted B-tree variant indexing structures. In Section 3, we first review the radix tree to help understand the main contributions of our work. Then, we present the three radix tree variants that we propose. We discuss the experimental environment in Section 4 and then present the experimental results in Section 5. Finally, we conclude with a summary in Section 6.

2 Background and Motivation

In this section, we review background work that we deem most relevant to our work and also necessary to understand our study. First, we review the consistency issue of indexing structures in persistent memory. Then, we present variants of B-trees for PM. As the contribution of our work starts with the radix tree, we review the radix tree in Section 3.

2.1 Consistency in Persistent Memory

Ensuring recovery correctness in persistent indexing structures requires additional memory write ordering constraints. In disk-based indexing, arbitrary changes to a volatile copy of a tree node in DRAM can be made without considering memory write ordering because it is a volatile copy and its persistent copy always exists in disk storage and is updated in disk block units. However, with failure-atomic write granularity of 8 bytes in PM, changes to an existing tree node must be carefully ordered to enforce consistency and recoverability. For example, the number of entries in a tree node must be increased after a new entry is stored. If the system fails after we increase the number of entries but before the new entry is stored in its corresponding space, the garbage entry previously stored in that space will be mistaken as a valid entry resulting in inconsistency.

In order to guarantee consistency between volatile CPU caches and non-volatile memory, we have to ensure the ordering of memory writes via memory fence and

cache line flush instructions. In the Intel x86 architecture, the CLFLUSH instruction is used to flush a dirty cache line back to memory and MFENCE is the load and store fence instruction that prevents the reordering of memory access instructions across the fence. Since CLFLUSH is ordered only with respect to MFENCE, CLFLUSH needs to be used along with MFENCE to prevent reordering of CLFLUSH instructions [21]. These memory fence and cache line flush instructions are known to be expensive [3, 20].

Another important aspect of maintaining consistency is the write size. In legacy B-tree variants, insertion or deletion of a node entry results in modification of a large portion of the node because the entries remain sorted. This is because insertion or deletion of an entry can result in shifts of data within the node. Such shifts are likely to be larger than the failure atomicity unit.

To resolve this problem, legacy systems generally rely on techniques such as logging or CoW. Logs can be used to undo or redo activities such that the system remains in a consistency state. CoW creates a copy and makes updates to the copy. This allows for atomic validation of the copy by overwriting the pointer with an atomic 8-byte store operation. Although logging and CoW guarantee consistency in the presence of failure, they hurt update performance especially when the updated data is large as they both need to duplicate the write operations.

2.2 Persistent B+-Trees

In recent years, several indexing trees for PM such as CDDS B-tree [17], NVTree [20], wB+Tree [3], and FP-Tree [15] have been proposed. To the best of our knowledge, all previously proposed persistent indexes are variants of the B-tree, which has been widely used in various domains including storage systems. We review each of these trees in detail below.

CDDS B-Tree: CDDS (Consistent and Durable Data Structure) B-tree is a multi-version B-tree (MVBT) for PM [17]. When a tree node is updated in the CDDS B-tree, it creates a copy of the updated entry with its version information instead of overwriting the entry, which guarantees recoverability and consistency. However, CDDS B-tree suffers from numerous dead entries and dead nodes. Also, it calls the expensive MFENCE and CLFLUSH instructions as many times as the number of entries in a tree node to sort the entries. Hence, CDDS B-tree is far from satisfactory in terms of both insertion and search performance.

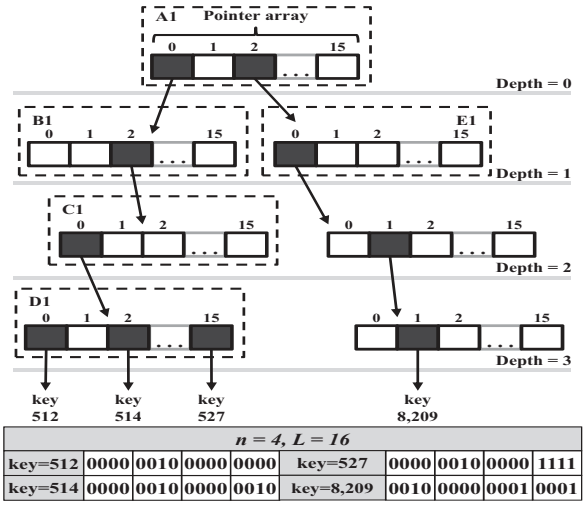
NVTree: NVTree proposed by Yang et al. [20] reduces the number of expensive memory fence and cache line flush instructions by employing an append-only update strategy. Due to this strategy and the fact that only the leaf nodes are kept in PM, NVTree requires only two cache line flushes, one for the entry and the other for the entry count, resulting in improved performance. This results in two consequences; first, the leaf node remains unsorted and second, the internal nodes may be lost upon system failure though the internal nodes can trivially be reconstructed using the leaf nodes in PM. However, NVTree requires all internal nodes to be stored in consecutive memory blocks to exploit cache locality, and within the large memory block, internal nodes are located by offsets instead of pointers. However, because NVTree requires internal nodes to be stored in consecutive blocks, every split of the parent of the leaf node results in the reconstruction of the entire internal nodes. We show in our experiments that due to this reconstruction overhead, NVTree does not perform well for applications that insert data on the fly.

FPTree: FPTree is another persistent index that keeps internal nodes in volatile memory while leaf nodes are kept in PM [15]. By storing the internal nodes in volatile memory, FPTree exploits hardware transactional memory to efficiently handle concurrency of internal node accesses. FPTree also proposes to reduce the cache miss ratio via fingerprinting. Fingerprints are one-byte hashes for keys in each leaf node. By scanning the fingerprints first before a query searches keys, FPTree reduces the number of key accesses and consequently, the cache miss ratio. Although FPTree shows superior performance to NVTree, FPTree also requires reconstruction of internal nodes when a system crashes.

wB+Tree: wB+Tree proposed by Chen and Jin also adopts an append-only update strategy, but unlike NVTree and FPTree, wB+Tree stores both internal and leaf nodes in PM [3]. Since the entries in internal nodes must be sorted, wB+Tree proposes to sort the entries via the slot array, which adds a level of indirection to the actual keys and pointers. That is, the slot array stores the index of keys in sorted order. Since the index is much smaller than the actual key and pointer, seven key indexes can be atomically updated via the 8-byte atomic write operation. wB+Tree also proposes to use an 8-byte bitmap to increase node capacity. When the bitmap is used, wB+Tree requires at least four cache line flushes. If only the slot array is used, the number of cache line flushes decreases to two. Although the number of cache line flushes is dramatically reduced compared to CDDS B-tree, wB+Tree carries the overhead of indirection. Also, wB+Tree still requires expensive logging or CoW for a node split.

3 Radix Trees for PM

In this section, we present the basics of the radix tree. We also discuss the three radix tree variants that we propose, namely, WORTPM (Write Optimal Radix Tree for PM) or simply, WORT (Write Optimal Radix Tree), WOART (Write Optimal Adaptive Radix Tree), and ART+CoW.



n = 4, L = 16									
key=512	0000	0010	0000	0000	key=527	0000	0010	0000	1111
key=514	0000	0010	0000	0010	key=8,209	0010	0000	0001	0001

Figure 1: An example radix tree

3.1 Radix Tree

Traditionally, radix trees come in two versions; one in basic form, which we refer to as the original radix tree, and the other that uses path compression to save memory [2, 26, 27, 28]. For ease of presentation, we first discuss the original version and defer the discussion on the path compressed version to Section 3.3.

The radix tree does not explicitly store keys in its tree nodes. Instead, a node consists of an array of child pointers, each of which is represented by a chunk of bits of a search key as in a hash-based index. Taking the radix tree example in Figure 1, each key is composed of 16 bits, with a chunk of 4 bits in length. Starting from the root node, the most significant leftmost 4 bits of each key is used to determine the subscript within the pointer array. Taking the key=527 case as a walking example, the left 4 bits, 0000, determines that the leftmost pointer points to the child node. In the next level, the next chunk of bits in the search key are used as the subscript of the child pointer array. This would be 0010 for key=527, meaning that the pointer in element 2 points to the next child. In this manner, in the walking example, we see that the next chunk of 4 bits, 0000, determines the next level child and that the least significant bits of the search key, 1111, are used in the leaf node.

There are two key characteristics of the radix tree that are different from B-tree variants. The first is that the height of the radix tree is determined and fixed by the length of the index key and the chunk size. For a maximum key length of L bits and the chunk, which represents the index to the child pointer, of n bits, which allows for a maximum 2^n child node pointers, the search path length is $\lceil L/n \rceil$. An often mentioned weakness of the radix tree is that its height ($\lceil L/n \rceil$) is, in general, taller than that of the B+-tree, which is $log_B N$, where N is the number of keys and B is the node degree [11]. This may result in deeper traversals of the tree.

The second characteristic is that the tree structure is independent of the insertion order but dependent on the distribution of keys. Whereas the B-tree variants maintain a balanced tree growing and shrinking according to the number of data, a radix tree has a fixed number of nodes determined by the maximum key range. How these nodes are used determines its effectiveness in terms of memory usage. For example, when the keys are sparsely distributed, the radix tree makes inefficient use of memory in contrast to when it is dense or skewed.

Due to these limitations and despite proposals to overcome such limitations [2], the radix tree has not been a popular indexing data structure. However, we find that the radix tree possesses features that may be exploited for efficient use with PM. First, with the radix tree, it is possible to traverse the tree structure without performing any key comparison because the positions of the child pointers are static and fixed according to its order. For example, if 527 is indexed as shown in Figure 1, 527 can be easily found by using each 4 bits as a subscript of the index without any comparison operation, i.e., `radix_index[0][2][0][15]`, as 527 is 0000 0010 0000 1111 in binary. In contrast, the B+-tree would require comparing the search key with other keys for each visited node. Such difference in activity can affect cache performance resulting in performance differences.

Also for insertion, the radix tree does not modify any existing entries for the same reason. That is, the number of child pointers in a radix tree node is fixed to 2^n, and it never overflows. Since the radix tree does not store keys, sorting, by nature, is not necessary. This is in contrast to B-tree variants that need to keep the keys sorted and also require expensive split or merge operations accompanied by logging to guarantee consistency.

3.2 Failure Atomic Write in Radix Tree

In this section, we describe the small changes that we made to make radix tree PM efficient. With the changes that we propose, the radix tree will remain consistent upon system failure without requiring any kind of logging or replication mechanism as long as the 8-byte failure atomicity assumption is satisfied. Essentially, there are just two simple changes that need to be made, which we describe in the following.

The first modification is making sure that a write to change a pointer is done in a particular order. Let us elaborate using an example of inserting key=3,884 (0000 1111 0010 1100 in binary) into the radix tree shown in Figure 1. With a given key, we traverse down the path using the partial keys until we find a pointer value that is NULL. For 3,884, as the first partial key is 0000, we follow the leftmost child pointer from the root node. At this level (depth 1), we find that the next partial key 1111 and

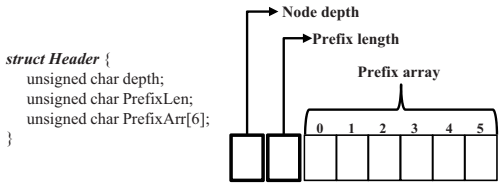


Figure 2: Compression header

that its pointer is NULL. Once we find the child pointer to be NULL, we create a new node and continue doing this until the leaf node is created. For the 3,884 example, we create a new node in depth 2 and a leaf node in depth 3. Finally, we replace the NULL pointers with the addresses of the new nodes.

The first modification that we propose is that the operation to replace the very first NULL pointer (if we have a sequence of nodes created) with the address of the next level node be the last operation. This ensures that the entire operation is failure-atomic. Since the pointer assignment statement is an 8-byte atomic write operation, no form of logging is necessary in the radix tree. However, we do need to call a few memory access serialization instructions to enforce failure atomicity. For example, if 8,209 is indexed as shown in Figure 1, first, we must call memory fence and cache line flush instructions to make sure all nodes leading to the leaf node, including the leaf node, is written to PM. Only then, should we change the very first NULL pointer in the root, that is, element 2 in node A1 to point to node E1. This change is then persisted with the memory fence and cache line flush instructions.

3.3 Radix Tree Path Compression

Thus far, we have described the original radix tree without path compression optimization. In this section, we describe the workings of path compression as our second proposed change is related to this matter.

Although the deterministic structure of the radix tree is the source of its good performance, it is also the weakest point as the key distribution has a high impact on the tree structure and memory utilization. If the distribution of the keys is sparse, the implicit key representation of the radix tree can waste excessive memory space. For example, suppose a string key is stored in a radix tree and there is no other key that shares the prefix with the key. If the number of child pointers in a node is 2^8, each node can implicitly index an 8-bit character but requires an 8-byte pointer per each child. That is, the tree node will use 8×256 bytes for each letter in the key. In order to mitigate this memory space utilization problem, we can consider reducing the number of child pointers in a node. However, this could result in a longer search path, possibly degrading search performance. The path compressed radix tree can save space by removing the internal nodes

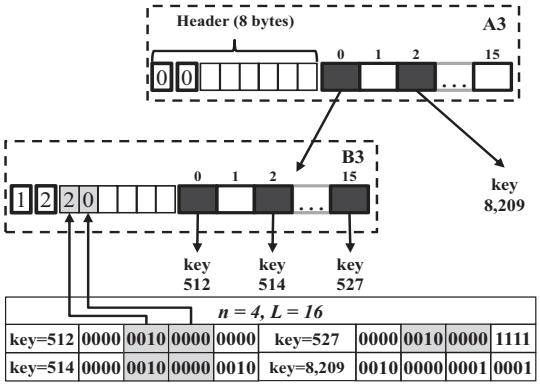


Figure 3: Path compression collapses radix tree nodes

that have only one child per node.

If a node in the radix tree has a single child such as key 8,209 in Figure 1, the node does not have to exist in order to distinguish it from other search paths. Hence, the node can be safely removed and created in a lazy manner until it becomes shared with another child node without hurting correctness. Path compression optimization in the radix tree truncates unique paths in the tree structure. Path compression is known to improve memory utilization especially when the key distribution is sparse. Moreover, path compression helps improve indexing performance by shortening the search path. There are three ways of implementing path compression in the radix tree, that is, the pessimistic, optimistic, and hybrid methods [11].

The pessimistic method explicitly saves the collapsed unique search path as the prefix array in the child node. The pessimistic method requires more memory space in the compressed nodes, but it can prune out unmatching keys instantly. On the other hand, the optimistic method stores the length of the collapsed prefix in the child node, instead of the collapsed prefix itself. Hence, the optimistic method cannot compare the collapsed prefix in the compressed node. Instead, it postpones the comparison of the collapsed keys until we reach a leaf node. The hybrid method combines the two by using the pessimistic method when the collapsed prefix is smaller than a specific length, and the optimistic method, otherwise.

Figure 2 illustrates the structure of a radix tree node header and Figure 3, which is a hybrid compressed version of Figure 1 as it simultaneously stores the length and the collapsed search path, shows how it is used to combine the inner nodes that share the common prefix. Prefix length specifies how many inner nodes are collapsed. In the example shown in Figure 3, 512, 514, and 527 share the second and third partial keys (2 and 0). Hence, the leaf node stores 2 as a prefix length. The prefix array is the collapsed common prefix. In the example, the prefix array stores the common prefix 2 (0010 in binary) and 0 (0000 in binary). Note that we always make use of an 8-

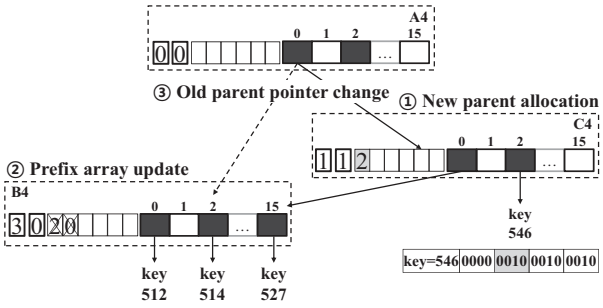


Figure 4: Path compression split

byte compression header. This is to maintain consistency with 8-byte atomic writes, which we elaborate on later.

Algorithm 1 SplitCmp(node,key,value,depth,diffPrfxIdx)

```
1: /*N=node,K=key,V=value,D=depth*/
2: Allocate newParent and newLeaf(K,V)
3: Move a part of header of N to header of newParent
4: Insert newLeaf and N into newParent as children
5: Allocate tmpHdr
6: Record header of N to be updated into tmpHdr
7: *((uint64*)&N.Hdr) = *((uint64*)&tmpHdr;
8: mfence();
9: clflush(&newLeaf);
10: clflush(&newParent);
11: clflush(&N.Hdr);
12: mfence();
13: /*Update old parent pointer*/
14: oldParent.children[]=newParent;
15: mfence();
16: clflush(&oldParent.children[]);
17: mfence();
```

3.4 WORT: Write Optimal Radix Tree

While it is trivial to make the radix tree that is not path compression optimized be failure-atomic as was shown in Section 3.2, doing so with the path compression optimized radix tree is more complicated as nodes are split and merged dynamically. The second modification that we propose makes the path compression optimized radix tree failure-atomic. (Hereafter, the radix tree we refer to are path compression optimized unless otherwise stated.) Note that even though the structure of the radix tree is no longer static, it is nevertheless still deterministic.

The second modification to the radix tree that we propose is the addition of the node depth information to each compression header, which was not needed in the legacy radix tree. Note that in our design, this requires one byte, which should be sufficient even in general deployment, and does not compromise the memory saving effect of the legacy path compression scheme. We now show how the node depth is used to guarantee failure atomicity during insertions. Let us again make use of an example.

Figure 4 is a depiction of how a compressed internal node (*B*3 in Figure 3) splits when the prefix of the key to insert does not match the prefix array, while Algorithm 1 describes the algorithm involved. Let us now go through the process step by step. Assume key=546 (0000 0010 0010 0010 in binary) is being inserted to the radix tree in Figure 3. Since 546 shares only the first partial key (4-bit prefix) with the prefix array of node *B*3, we need to split and create a parent node for the smaller common prefix. This new node is depicted as *C*4 in Figure 4. Once the new node *C*4 stores child pointers to node *B*4 and key 546, node *B*3 (in Figure 3) needs to delete the two partial keys in the prefix array (lines 5-12 in Algorithm 1). Also, node *A*3 (in Figure 3) needs to replace the child pointer to *B*3 to point to *C*4 (lines 12-17) as shown in node *A*4, and this must be done atomically. Otherwise, failure atomicity is not guaranteed and the tree may end up in an inconsistent state. For example, say the system crashes as *B*3 is changed to *B*4, but *A*3 is still unchanged.

Since multiple tree nodes cannot be updated atomically, persistent structures such as the B-tree variants employ expensive logging methods. However, we find that the radix tree can tolerate this temporary inconsistent state by storing the depth in each node and skipping the comparison of the partial key that is currently being expanded to a new parent node. That is, if the node that expands can atomically update its depth, prefix length, and prefix array altogether, then the radix tree can return to its consistent state without relying on logging.

Consider the example of Figure 4 once again. Before node *B*3 is expanded, the depth of node *B*3 is 1 (starting from 0 at root node) and the prefix length is 2, which indicates that node *B*3 indexes the second and third prefixes. After creating node *C*4, the depth and the prefix length need to be updated to 3 and 0, respectively. Assume the system crashes after we update the prefix length and depth, but before we update the child pointer of node *A*3. If we only had the prefix length as in the tradition radix tree, there is no way to detect node *B*4 is in an inconsistent state. However, if the depth is atomically updated along with the prefix length, which is possible with the 8-byte failure atomic write assumption, we can easily detect that there is a missing node between *A*3 and *B*4. Once we detect the inconsistency, the inconsistent node can reconstruct its previous depth, prefix length, and prefix array by selecting two leaf nodes from two arbitrary search paths (lines 3-4 of Algorithm 2) and recomputing the common prefix (lines 5-12 of Algorithm 2). Note that path compression guarantees the existence of at least two leaf nodes in any internal node and that the prefix array has the largest common prefix of every key in a node. In the example of Figure 4, suppose node *B*4 selects key=512 (0000 0010 0000 0000 in binary) and key=527 (0000 0010 0000 1111 in binary) as the two arbitrary leaf nodes. The largest common prefix of those two keys

Algorithm 2 RecoverHeader(node,depth)

```
 1: /*N=node,D=depth*/
 2: /*Select two different arbitrary leaves of N*/
 3: L1 = SelectArbitraryLeaf(N);
 4: L2 = SelectArbitraryLeaf(N);
 5: safeHdr = AllocHeader();
 6: safeHdr.depth=D;
 7: Compute the largest common prefix of L1 and L2
 8: Record it into prfxLen and prfxArr of safeHdr
 9: *((uint64*)&N.Hdr) = *((uint64*)&safeHdr);
10: mfence();
11: clflush(&N.Hdr);
12: mfence();
```

is 0000 0010 0000 in binary. The first prefix 0000 is ignored because we reconstruct the node in depth 1.

We name the radix tree that incorporates the two modifications that we mentioned, one in Section 3.2 and one in this section, WORTPM for Write Optimal Radix Tree for PM, or just WORT. WORT is optimal in that consistency is accomplished by using only 8-byte failure-atomic writes for every operation without requiring any logging or duplication of data under any circumstance.

3.5 Write Optimal Adaptive Radix Tree

Even with path compression, with the radix tree, there is a well known trade-off between tree traversal performance and memory consumption, i.e., if we increase the number of child pointers in a node, the tree height decreases but node utilization is sacrificed. Poor node utilization and high memory consumption have been pointed out as the major disadvantages of the radix tree. In order to resolve these problems, studies such as Generalized Prefix Tree [2] and Adaptive Radix Tree (ART) [11] have been conducted. In this section, we discuss ART and how we adapt ART for PM.

ART is a space efficient radix tree that adaptively changes its node size according to node utilization. In order to reduce the tree height, ART sets the number of child pointers in a node to 2^8 and uses one-byte sized partial keys per node. In parallel, ART reduces memory space consumption by using one of four different node types, namely, NODE4, NODE16, NODE48, and NODE256, according to node utilization. Starting with NODE4, ART adaptively converts nodes into larger or smaller types as the the number of entries exceeds or falls behind the capacity of a node type. Although ART has been shown to outperform other state-of-the-art cache conscious in-memory indexing structures including FAST [11], ART in its current form does not guarantee failure atomicity in PM. Hence, we redesign the node structure of ART, without compromising its memory saving effect, to enforce failure atomicity in PM, which we refer to as Write Optimal Adaptive Radix Tree (WOART). In particular, we find that for PM, NODE4

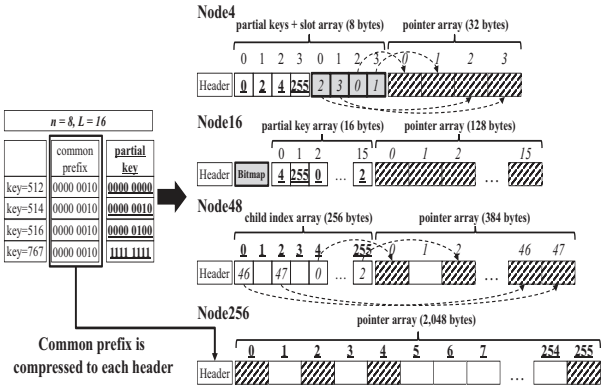


Figure 5: Node structures in WOART

and NODE16 have to be redesigned, NODE48 slightly modified, and NODE256 left unchanged from the original design.

NODE4: In order to reduce the node size when node utilization is low, ART utilizes the node type NODE4, a node that has no more than 4 entries. Since implicit key representation in NODE4 wastes memory space and is not helpful in distinguishing index keys, NODE4 explicitly stores partial keys. Hence, in the original ART scheme, four pairs of partial keys and pointers are kept. The partial keys and pointers are stored at the same index position in parallel arrays and sorted together by the partial key values. With sorting employed, some form of logging becomes a requirement to ensure consistency.

In WOART, we make the following changes to ensure consistency with a single 8-byte atomic write. First, pointers are updated in append-only manner only when an empty entry is available. Then, we add a level of indirection by having a separate slot array that serves as an index to the position of the pointer corresponding to the partial key as shown in NODE4 of Figure 5. Finally, we make use of the fact that the partial key size in the radix tree node is just one byte and that NODE4 only stores four keys per node, plus the fact that the entire slot array size is also only four bytes. Hence, the partial keys and the slot array in NODE4 altogether can be written in a single 8-byte atomic write operation. By performing this operation as the last update, consistency of the tree is guaranteed.

Algorithm 3 shows the details of the insertion algorithm for NODE4. Deletion is omitted as it is analogous to insertion. First, we look for an empty pointer in the slot array (line 1). If there is one (idx is returned), we first make a copy of the 8-byte partial keys and slot array and insert the partial key value and the idx value into the copy (lines 3 and 4). Then, we store the child address in the pointer array entry indexed by idx, and call `mfence` and `clflush` (lines 5-8). Finally, we atomically update the partial keys and slot array by atomically overwriting

Algorithm 3 AddChild4(node4,PartialKey,child)

```
1:  if node4 is not full then
2:      idx = getEmptyIdx(node4.slot);
3:      CopySlot(tmpSlot,node4.slot);
4:      InsertKeytoSlot(tmpSlot,PartialKey,idx);
5:      node4.ptrArr[idx]=child;
6:      mfence();
7:      clflush(&node4.ptrArr[idx]);
8:      mfence();
9:      *((uint64*)node4.slot)=*((uint64*)&tmpSlot);
10:     mfence();
11:     clflush(node4.slot);
12:     mfence();
13: else
14:     Copy and Exchange node4 to node16
```

Algorithm 4 AddChild16(node16,PartialKey,child)

```
1:  if node16 is not full then
2:      idx=getEmptyIdx(node16.bitmap);
3:      node16.partialkeys[idx]=PartialKey;
4:      node16.ptrArr[idx]=child;
5:      mfence();
6:      clflush(&node16.partialkeys[idx]);
7:      clflush(&node16.ptrArr[idx]);
8:      mfence();
9:      node16.bitmap+ = (0x1UL << idx);
10:     mfence();
11:     clflush(&node16.bitmap);
12:     mfence();
13: else
14:     Copy and Exchange node16 to node48
```

the original with the copy of the 8-byte partial keys and slot array and call `mfence` and `clflush` (lines 9-12).

If a node needs more than 4 entries, we expand the node into a NODE16 type through memory allocation and multiple write operations. Note, however, that consistency is always guaranteed during this expansion as the final change to the parent of the new NODE16 is always done with a single 8-byte atomic write.

NODE16: A NODE4 type node becomes a NODE16 type when the number of child pointers grows past four and can have as many as 16 child pointers. Similarly to NODE4, the original NODE16 node keeps the partial keys sorted. However, as previously mentioned, this is an expensive task with PM.

In WOART, with NODE16, we take a similar approach as with NODE4 as we explicitly store keys in append-only manner. However, unlike NODE4, NODE16 does not sort partial keys nor use a slot array to store the indexes to child pointers. Instead, partial keys and pointers are stored as parallel arrays, denoted as partial key array and pointer array, respectively, in Figure 5. Note that there is also a 16-bit bitmap that distinguishes the valid and invalid key and pointer values. The atomic write of this bitmap ensures the consistency of the tree.

Specifically, and in relation to Algorithm 4, we refer to the bitmap (line 2) to find an empty entry to insert the partial key. Then, the partial key and pointer values are placed in the empty entry position of the parallel arrays (lines 3-8). Finally, the bitmap position of the empty entry is set to 1 with an atomic write (lines 9-12). This guarantees consistency of the radix tree.

Note that for deletion, we need to manipulate the bitmap to indicate the invalidness of partial keys and pointers. This is also an atomic write operation, hence, consistency is maintained. Also note that although the partial keys are not sorted, this does not hurt search performance as NODE16 has no more than 16 partial keys. Comparing a given search key with 16 partial keys that fit in a single cache line can be performed very efficiently in modern processors. As this is not true when the number of keys becomes large, WOART explicitly stores partial keys only for NODE4 and NODE16 types.

NODE48 and NODE256: Let us now go over the NODE48 and NODE256 type nodes in WOART. As shown in Figure 5, a NODE256 type node is exactly the same as a node in the original radix tree (see Figure 1). Hence, for NODE256, we simply make use of WORT and do not discuss NODE256 any further.

For NODE48, the details are referred to the original ART [11] as it is essentially what is used. We make use of the open source code provided by Leis et al. [24] but do make slight code changes to make it consistent in PM. However, the essence is the same. Specifically, NODE48 keeps two separate arrays, one that has 256 entries indexed by the partial key and one that has 48 entries, each of which will hold one of the child pointers, respectively denoted child index array and pointer array in Figure 5. Consistency is ensured by making writes to the pointer array first, and then atomically writing the pointer array index value to the child index array, the index of which is determined by the partial key. Note that we search for an available entry in the pointer array by checking the child index array. If we instead search for an available entry by checking for a NULL pointer in the pointer array, as was implemented by Leis et al. [24], system failure may result in leaving non-reusable invalid pointers.

3.6 ART with Copy-on-Write

The third radix tree variant that we consider in this study is one that makes use of copy-on-write (CoW). CoW in the radix tree is much simpler than that with B-tree variants as CoW occurs for only one node upon an update as only the updated node itself is affected upon an update. That is, for any update one can simply create a copy of the node and maintain consistency by replacing the pointer in its parent node with the address of the copy at the final moment. This is in contrast to B-tree variants where node changes can cascade to other nodes, for example, due to splits forcing parent nodes to also split.

In this study, we consider ART with CoW, which we

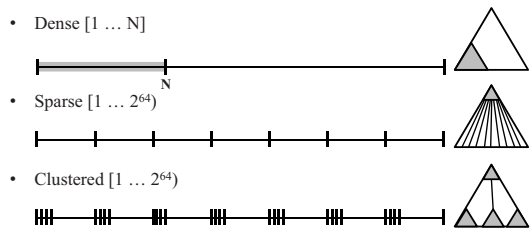


Figure 6: Workloads

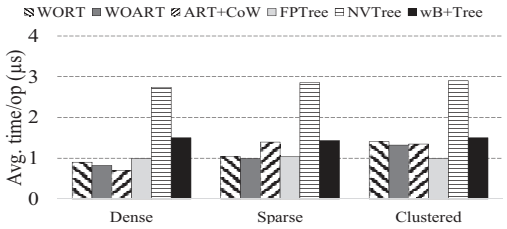


Figure 7: Insertion performance in DRAM latency

refer to as ART+CoW. ART+CoW combines the features of WORT, WOART, and CoW. First, for NODE256, we incorporate the first modification of WORT discussed in Section 3.4. However, for path compression, instead of adding the node depth, we use CoW for both split and merge. Second, for NODE48, we make use of the same mechanism as in WOART. Finally, for NODE16 and NODE4, we simply employ CoW. That is, we make a copy of the node, make changes to it, then change the parent pointer value with the 8-byte failure-atomic write.

4 Experimental Environment

To test the effectiveness of the proposed radix trees, we implement them and compare their performance with state-of-the-art PM indexing structures. The experiments are run on a workstation with an Intel Xeon E5-2620 v3 2.40GHz X 2, 15MB LLC (Last Level Cache), and 256GB DRAM running the Linux kernel version 4.7.0. We compile all implementations using GCC-4.4.7 with the -O3 option.

To observe the effect of PM latency on the performance of the data structure, we emulate PM latency using Quartz [1, 18], a DRAM-based PM performance emulator. Quartz emulates PM latency by injecting software delays per each epoch and throttling the bandwidth of remote DRAM using thermal control registers. We emulate read latency of PM using Quartz while disabling its bandwidth emulation. Since write memory latency emulation is not yet supported in the publicly available Quartz implementation [1], we emulate PM write latency by introducing an additional delay after each `clflush` and `mfence` instructions, as in previous studies [7, 9, 19]. No delays are added for the `store` instruction as the CPU cache hides such delays [25].

For comparison, we implement wB+Tree, NVTree and FPTree [3, 15, 20]. For wB+Tree, we implement both the *slot-only* and *bitmap+slot* schemes, but we present the performance of only the bitmap+slot scheme denoted as `wB+Tree` because we observe that the bitmap+slot scheme has lower node split overhead and search performance is better due to the large node degree. Note that the internal nodes of NVTree and FPTree are designed to be volatile and does not guarantee failure atomicity. As we consider PM latency in our experiments, for NVTree and FPTree, we distinguish latency for internal nodes in DRAM and leaf nodes in PM.

For the workloads, we make use of three synthetically generated distributions of 8-byte integers. Unlike B-tree based indexes, the radix tree is sensitive to the key distribution due to its deterministic nature. To see how the indexes react to extreme cases, we consider three distributions as shown in Figure 6. In Dense key distribution, we generate sequential numbers from 1 to 128M, so that all keys share a common prefix. This workload is the ideal case for the radix tree since overall node utilization is 100%. In Sparse key distribution, keys are uniformly distributed, thus they share a common prefix only in the upper level of the tree structure. For the lower level nodes, the radix tree relies on path compression optimization to improve node utilization. In Clustered key distribution, we merge Dense and Sparse key distributions to model a more realistic workload. Specifically, we generate 2 million small dense distributions, each consisting of 64 sequential keys. In Clustered key distribution, the middle level nodes share common prefixes. For all three distributions, the keys are inserted in random order and experimental results are presented by using a single thread.

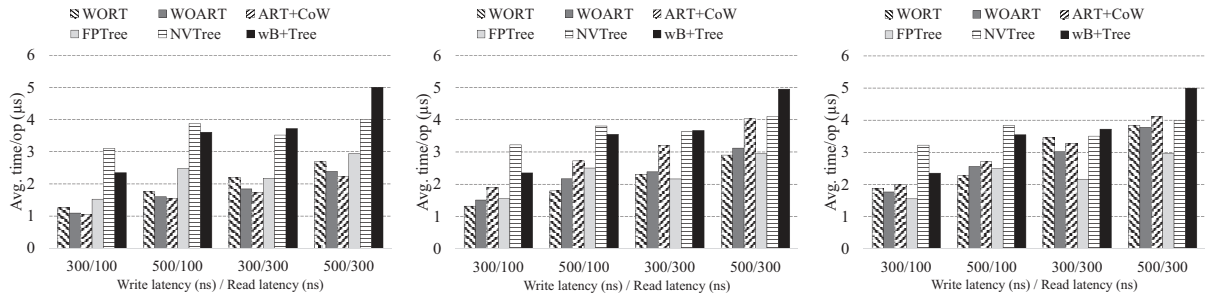
5 Performance Evaluation

In this section, we evaluate the three proposed radix tree variants against the state-of-the-art persistent indexing structures, namely, wB+Tree, NVTree, and FPTree.

5.1 Insertion Performance

Figure 7 shows the average insertion time for inserting 128 million keys for the three different distributions when the entire memory space is of DRAM latency. We set the number of child pointers of WORT to 2^4 so that each node indexes 4-bit partial keys for a maximum of 16 child pointers. We see from the results that in general the radix based trees perform considerably better than NVTree and wB+Tree. The range of benefits and the best radix tree variant depends on the workload.

For NVTree, performance suffers because it requires internal nodes to be pre-allocated in consecutive memory space and a node split results in reconstruction of all the internal nodes. FPTree performs the best among the B-tree variants and, in some cases, better than the radix



(a) Dense (b) Sparse (c) Clustered

Figure 8: Insertion Performance Comparison

Table 1: Average LLC Miss and CLFLUSH per Insertion, LLC Miss per Search, and Leaf Node Depth for Various PM Indexing Structures

D (Dense) **S** (Sparse) **C** (Clustered)	**(a) Average number of LLC miss per insertion**			**(b) Average number of CLFLUSH per insertion**			**(c) Average number of LLC miss per search**			**(d) Average leaf node depth**		
	D	**S**	**C**	**D**	**S**	**C**	**D**	**S**	**C**	**D**	**S**	**C**
WORT	6.3	7.2	17.0	2.2	2.4	2.3	6.5	7.6	11.5	7.0	7.0	8.2
WOART	5.7	7.9	11.2	2.4	3.5	3.7	4.8	7.9	8.9	4.0	4.0	4.2
ART+CoW	5.0	12.7	12.9	2.4	3.8	3.9	3.8	6.2	8.8	4.0	4.0	4.2
FPTree	6.8	6.8	6.8	4.8	4.8	4.8	13.9	14.1	14.1	3.0	3.0	3.0
NVTree	35.0	35.6	33.6	3.3	3.3	3.3	33.5	33.5	33.3	3.0	3.0	3.0
wB+Tree	22.3	22.4	22.4	6.0	6.0	6.0	22.9	22.8	23.3	4.0	4.0	4.0

tree variants. However, this comparison must be made with caution as FPTree assumes that the internal nodes are in DRAM. This has the drawback that when the system recovers from failure or rebooted the internal nodes must be reconstructed incurring considerable overhead.

Considering only the radix trees for the distributions in Figure 7, we see that for Clustered distribution, insertion time is roughly 1.5× higher than for the other two. As shown in column (a) in Table 1, this is due to the higher number of LLC misses incurred as the common prefix of the Clustered distribution is much more fragmented due to the scattered tree nodes than the other two distributions.

Figure 8 shows the insertion results as the latency for reads and writes are changed. The numbers on the *x*-axis represent the latency values in nanoseconds. The default latency, that is of DRAM as reported by Quartz, is 100ns. As PM read and write latency is generally expected to be comparable or slightly worse than those of DRAM, we set the latency to various values as shown in the figure. For these experiments, the internal nodes of NVTree and FPTree are considered to be in DRAM, hence not affected by the latency increase of PM. This should result in more favorable performance for these two trees.

Throughout the results, whether read or write latency is increased, we see that the radix tree variants consistently outperform the B-tree variants, except for FPTree. However, as latency increases, wB+Tree and the radix tree variants that store every node in PM suffer more.

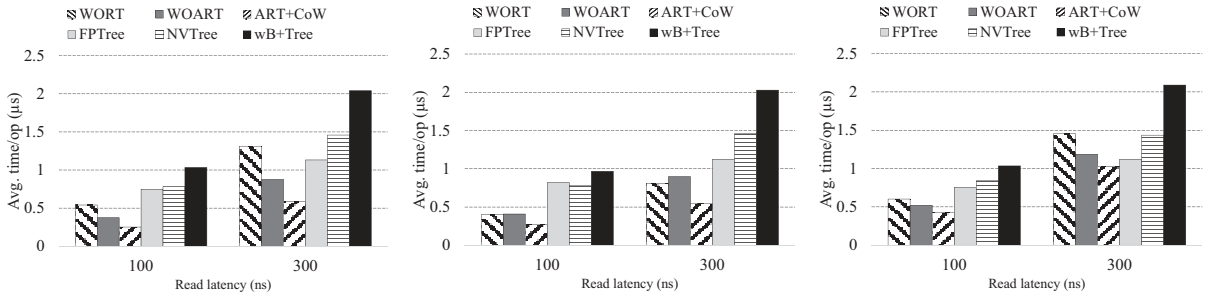
We also see that the B-tree variants are, in general, more sensitive to write latency increases. Column (b) in Table 1, which is the measured average number of cache line flushes per insertion, shows the reason behind this. We see that B-tree variants incur more cache flush instructions than the radix tree variants.

5.2 Search Performance

Figure 9 shows the average search time for searching 128 million keys for the three different distributions. Since search performance is not affected by the write latency of PM, we vary only the read latency using Quartz.

First, observe the left part of each graph, where both read and write latencies of PM are the same as DRAM. We see that the radix tree variants always perform better than the B-tree variants. In particular, ART+CoW performs the best for all cases. Since ART+CoW uses copy-on-write to ensure consistency, there is no additional indirection caused by the append-only strategy and the alignment of partial keys can be maintained. Therefore, ART+CoW is advantageous in tree searching compared to WOART where additional indirection and unsorted keys are employed to support the append-only strategy.

The reason that the radix tree variants perform better can be found in columns (c) and (d) in Table 1 that shows the average number of LLC misses and the average leaf node depth, respectively. We see that the overall performance is inversely proportional to the number of LLC misses and the depth of the tree. Notice that the depth of the tree is slightly higher for the radix tree variants. However, the number of LLC misses is substan-



(a) Dense (b) Sparse (c) Clustered

Figure 9: Search Performance Comparison

tially smaller, which compensates for the higher depth. The reason there are fewer LLC misses is because the radix tree can traverse the tree structure without performing any key comparisons, which incurs less pollution of the cache. Recall that, in contrast, B-tree variants must compare the keys to traverse the tree and that the keys may even be scattered across the entire node due to the append-only strategy. Hence, the B-tree variants more frequently access the entire range of the node causing more LLC misses.

Now, consider the right part of each graph, which is when read latency is increased to 300, in comparison with the left part. We see that WORT stands out in latency increase especially for Dense and Clustered workloads. This is due to the depth of the tree as WORT has the highest depth. Other than WORT, we see that WOART and ART+CoW perform better than the B-tree variants even with the increased read latency even though the internal nodes of FPTree and NVTree are still seeing DRAM latency.

5.3 Range Query Performance

Traditionally, B+-trees have an edge over radix trees on range queries as keys are sorted within the nodes and the leaf nodes are linked with sibling pointers. In contrast, in the radix tree, no sibling pointers exist, but the leaf nodes essentially contain keys that are implicitly sorted. Hence, in the radix tree, one may have to traverse up the descendant node(s) in order to find the next leaf node.

Our proposed radix tree variants are no different from traditional radix trees and do not do well for range queries. However, we note that the B-tree variants for PM also do not keep the keys sorted to reduce the overhead for ensuring consistency, which harms one of the key features of B+-trees. To see the effect of such changes we perform experiments for range queries.

Figure 10 shows the range query performance when keys in the range consisting of 0.001% and 0.01% of the 128M keys are queried. Here, we also present the performance of the original B+-tree for reference. We observe that the average time per operation of the three radix tree variants is over 5.8× and 6.4× than B+-tree for the 0.001% and 0.01% range, respectively. However, the performance gap declines for PM indexes. With respect to FPTree, NVTree, and wB+Tree, the average time per operation of the three radix variants is 3.0× and 2.8×, 1.8× and 1.8×, and 4.8× and 5.3× higher for 0.001% and 0.01% range queries, respectively. The reduction in difference is because B-tree variants need to rearrange the keys when servicing range queries.

5.4 Experiments with Memcached

In order to observe the performance of our proposed index structures for real life workloads, we implement all the tree structures used in the previous experiments within Memcached. Memcached is an in-memory caching system for key-value based database systems [22]. We remove the hash function and table of Memcached and embed the indexing structures. We also replace the bucket locking mechanism of the hash table with the global tree locking mechanism. The global tree locking mechanism locks the root of the tree in order to prevent conflicts between threads whenever an insertion operation is executed. We run mc-benchmark, which performs a series of insert queries (SET) followed by a series of search queries (GET) [23]. The key distribution is uniform, which randomly chooses a key from a set of string keys.

For these experiments, we use two connected machines with a 10Gbps Ethernet switch, one for Memcached and the other for running the mc-benchmark. We execute 128 million SET and GET queries with 4 threads and 50 threads, respectively. The machine used for Memcached is the same as described in Section 4 and the machine that runs the mc-benchmark is an Intel i7-4790 3.60GHz, 32GB DRAM and with Linux version 4.7.0.

For these experiments, all indexing structures are assumed to run entirely on PM even for the internal nodes of FPTree. This is because to consider the hybrid configuration, considerable changes must be made to Memcached, which we wanted to avoid as such changes may affect the outcome of the results. To support variable-sized string keys, wB+Tree and FPTree replace the keys in nodes with 8-byte pointers to the keys stored in a sep-

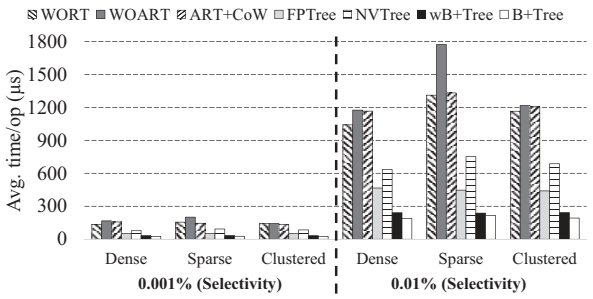


Figure 10: Range query over 128M

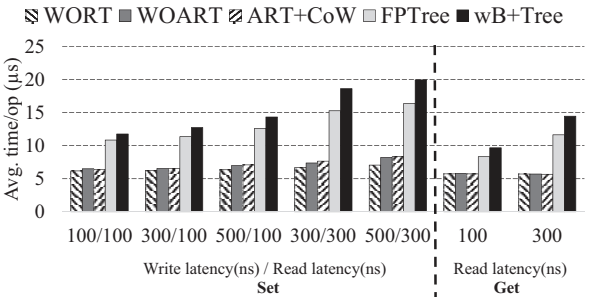


Figure 11: Memcached mc-benchmark performance

arate location [3, 15]. We follow this design in our experiments. For NVTree, as there is no mention on how to handle variable-sized keys, we omit its evaluation for Memcached [20].

The left part of Figure 11 shows the results for SET operations. We observe that the radix tree variants perform considerable better than the B-tree variants by roughly 50%. Other than the difference in structure, there are a couple of other factors that influences the difference. One is that there is the additional indirection and large key-comparing overhead, which was also observed by Chen and Jin [3]. Note that for radix tree variants, the overhead for key comparison is minimal. The other is the additional cache line flush required to store the keys in a separate PM area in the case of B-tree variants. This overhead does not exist for radix tree as variable-sized strings can be handled in essentially the same manner as integers. We also see that the effect of increased PM latency is also more profound for the B-tree variants.

The right part of Figure 11 shows the results for GET queries. Similarly to the SET query, the radix tree variants perform better than the B-tree variants. However, we also notice that the radix tree variants' results are the same for both 100 and 300 read latencies. We conjecture that this actually represents the network communication bottleneck. In spite of this, however, we see that the radix tree variants reduce wB+Tree latency by 41% and 60% and FPTree latency by 31% and 51% for 100 and 300 read latencies, respectively.

6 Summary and Conclusion

With the advent of persistent memory (PM), several persistent B-tree based indexing structures such as NVTree [20], wB+Tree [3], and FPTree [15] have recently been proposed. While B-tree based structures have been popular in-memory index structures, there is another such structure, namely, the radix tree, that has been less popular. In this paper, as our first contribution, we showed that the radix tree can be more appropriate as an indexing structure for PM. This is because its structure is determined by the prefix of the inserted keys dismissing the need for key comparisons and tree rebalancing. However, we also show that the radix tree as-is cannot be used in PM.

As our second contribution, we presented three radix tree variants adapted for PM. WORT (Write Optimal Radix Tree), which is the first variant that we proposed, employs a failure-atomic path compression scheme that we develop. WORT is optimal for PM, as is WOART, in the sense that they only require one 8-byte failure-atomic write per update to guarantee the consistency of the structure totally eliminating the need to make duplicates typically done via logging or copy-on-write (CoW) in traditional structures. WOART (Write Optimal Adaptive Radix Tree) and ART+CoW, the second and third variants, are both based on the Adaptive Radix Tree (ART) that was proposed by Leis et al. [11]. ART resolves the trade-off between search performance and node utilization found in traditional radix trees by employing an adaptive node type conversion scheme that dynamically changes the size of a tree node based on node utilization. However, ART in its present form does not guarantee failure atomicity. WOART redesigns the adaptive node types of ART and supplements memory barriers and cache line flush instructions to prevent processors from reordering memory writes and violating failure atomicity. ART+CoW, the third variant, extends ART to make use of CoW to maintain consistency.

Extensive performance studies showed that our proposed radix tree variants perform considerable better than recently proposed B-tree variants for PM such as NVTree, wB+Tree, and FPTree for synthetic workloads as well as in implementations within Memcached.

Acknowledgement

We would like to thank our shepherd Vasily Tarasov and the anonymous reviewers for their invaluable comments. We also thank our colleagues from the NECSST lab and Wook-Hee Kim for their numerous discussions that helped shape this research. We are also grateful to Ismail Oukid for his feedback in implementing the FPTree. This work was supported by Samsung Research Funding Centre of Samsung Electronics under Project Number SRFC-IT1402-09.

References

[1] GitHub. Quartz: A DRAM-based performance emulator for NVM *https://github.com/HewlettPackard/quartz*.

[2] M. Boehm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient In-Memory Indexing with Generalized Prefix Trees. In *Proceedings of the Database Systems for Business, Technology, and Web (BTW)*, 2011.

[3] S. Chen and Q. Jin. Persistent B+-Trees in Non-Volatile Main Memory. In *Proceedings of the VLDB Endowment (PVLDB)*, 2015.

[4] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[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] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)*, 2014.

[7] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware Logging in Transaction Systems. In *Proceedings of the VLDB Endowment (PVLDB)*, 2014.

[8] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.

[9] W. -H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, 2016.

[10] E. Lee, H. Bahn, and S. 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.

[11] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the IEEE 29th International Conference on Data Engineering (ICDE)*, 2013.

[12] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *Proceedings of the ACM Conference on Timely Results in Operating Systems (TRIOS)*, 2013.

[13] D. Narayanan and O. Hodson. Whole-System Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[14] J. Ou, J. Shu, and Y. Lu. A High Performance File System for Non-Volatile Main Memory. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, 2016.

[15] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.

[16] J. Rao and K. 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.

[17] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[18] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 15th Annual Middleware Conference (Middleware)*, 2015.

[19] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[20] J. Yang, Q. Wei, C. Chen, C. Wang, and K. L. Yong. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[21] Intel Corporation Intel® 64 and IA-32 Architectures Software Developer's Manual. *http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html*.

[22] MEMCACHED What is Memcached? *https://memcached.org*.

[23] GitHub Memcache port of Redis benchmark. *https://github.com/antirez/mc-benchmark*.

[24] Technische Universität München The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. *https://db.in.tum.de/ leis/*.

[25] J. Hyun Kim, Young Je Moon, and Sam H. Noh. An Experimental Study on the Effect of Asymmetric Memory Latency of New Memory on Application Performance. In *Proceedings of the 24th IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2016.

[26] J. Corbet Trees I: Radix trees. *https://lwn.net/Articles/175432/*.

[27] D. R. Morrison PATRICIA –Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM (JACM)*, 1968.

[28] D. E. Knuth The Art in Computer Programming: Sorting and Searching, Vol. 3. *Pearson Education*, 1998.