2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

HART: A Concurrent Hash-Assisted Radix Tree for DRAM-PM Hybrid Memory Systems

Wen Pan
Computational Science Research Center
San Diego State University
San Diego, USA
wpan@sdsu.edu

Tao Xie
Department of Computer Science
San Diego State University
San Diego, USA
txie@sdsu.edu

Xiaojia Song
Computational Science Research Center
San Diego State University
San Diego, USA
xsong2@sdsu.edu

***Abstract*—Persistent memory (PM) exhibits a huge potential to provide applications with a hybrid memory system where both DRAM and PM are directly connected to a CPU. In such a system, an efficient indexing data structure such as a persistent tree becomes an indispensable component. Designing a capable persistent tree, however, is challenging as it has to ensure consistency, persistence, and scalability without substantially degrading performance. Besides, it needs to prevent persistent memory leaks. While hash table has been widely used for main memory indexing due to its superior performance in random query, ART (Adaptive Radix Tree) is inherently better than B/B$^+$-tree in most basic operations on both DRAM and PM. To exploit their complementary merits, in this paper we propose a novel concurrent and persistent tree called HART (Hash-assisted ART), which employs a hash table to manage ARTs. HART employs a selective consistency/persistence mechanism and an enhanced persistent memory allocator, which can not only optimize its performance but also prevent persistent memory leaks. Experimental results show that in most cases HART significantly outperforms WOART and FPTree, two state-of-the-art persistent trees. Also, it scales well in concurrent scenarios.**

***Index Terms*—ART, hash table, persistent tree, selective consistency/persistence, concurrent access, persistent memory leak**

I. Introduction

Persistent memory (PM, also called storage class memory or SCM) is a byte-addressable non-volatile memory (NVM), which can be directly connected to the main memory bus and accessed by CPU through load/store instructions [1]. Emerging PM technologies include 3D XPoint [2], phase change memory (PCM), spin-transfer torque memory (STT-RAM), resistive memory (RRAM), FeRAM, and memristor. Their near-DRAM performance plus DRAM-like byte-addressability as well as disk-like persistence and capacity inspire a DRAM-PM hybrid memory system where PM is connected directly to a CPU [3]. In such a system, most modules of an operating system are still running on DRAM while a PM-oriented file system (e.g., PMFS [1] or NOVA [4]) or a key-value store (e.g., HiKV [5]) managing user data on a PM device. Obviously, an efficient persistent indexing data structure such as a persistent indexing tree (hereafter, persistent tree) becomes indispensable.

This work is sponsored by the U.S. National Science Foundation under grant CNS-1813485.

Designing a capable persistent tree, however, is challenging [6]. Unlike its volatile counterpart, a persistent tree has to ensure data consistency when a system failure occurs. Otherwise, if an update is being carried out on a data structure in PM when a system crash happens, it could be left in a corrupted state. Thus, a data consistency mechanism is required to solve this problem. Unfortunately, prior studies found that most data consistency mechanisms like logging or copy-on-write (CoW) incur a significant performance penalty [6], [7]. Developing a persistent tree that can ensure data consistency without substantially degrading performance becomes a challenge. Besides, a persistent tree needs to prevent persistent memory leaks [8]. Memory leaks are more detrimental for PM than for DRAM because they are persistent in PM [8].

Several persistent trees [3], [6]–[9] have been proposed recently. Most of them are a variant of either a B-tree [3] or a B$^+$-tree [6], [8], [9]. Lee *et al.*, however, proposed three persistent trees that are variants of a radix tree [7]. In fact, a radix tree is more appropriate to PM due to some of its unique features. For example, the height of a radix tree depends on the length of the keys rather than the number of records [10]. Besides, it does not demand tree re-balancing operations and node granularity updates [10]. We found that existing B/B$^+$-tree based persistent trees [3], [6], [8], [9] face two dilemmas. First, they are in a dilemma in deciding whether or not to keep nodes sorted. In a traditional B/B$^+$-tree, all keys in a node are sorted so that a search operation can be quickly carried out. However, the overhead of keeping nodes sorted in PM is high. Therefore, NV-Tree [6] and FPTree (Fingerprinting Persistent Tree) [8] choose to leave nodes unsorted in order to avoid that overhead. As a result, their search performance is greatly degraded. Since each leaf node in a radix tree only contains one record, the dilemma does not exist in a radix tree. Second, they are in a dilemma in whether or not to still maintain tree balance, which is a critical property of a B/B$^+$-tree. An unbalanced B/B$^+$-tree not only yields a degraded performance but also wastes space. However, performing a tree re-balancing operation such as a merge operation (i.e., two nodes with each having keys no more than half of its capacity are merged into one new node) in PM causes a noticeable cost. A radix tree has no such issue.

In addition to various trees, hash table is another widely

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00100

used indexing data structure in main memory. Without hash collisions, the time complexity of a search/insertion operation is $O(1)$. In contrast, the time complexity of these operations is $O(h)$ for a tree structure where h is the height of the tree. Therefore, compared with B$^+$-trees and radix trees, hash table can deliver better search performance for sparse keys [10]. However, hash table has its own limitations. First, since a hash table scatters the keys through a hash function, its range query performance is much worse than that of a tree. Second, the scalability of a hash table is not as good as that of a tree. When the number of records grows hash collision happens more frequently, which is detrimental to its performance. Third, its insertion performance is worse than that of a radix tree under various workloads. To exploit the complementary merits of a radix tree and a hash table, in this paper we propose a novel concurrent and persistent tree called HART (Hash-assisted Adaptive Radix Tree), which utilizes a hash table to manage multiple adaptive radix trees (ARTs).

HART only stores the leaf nodes of ARTs in PM. The hash table and the internal nodes of ARTs are all stored in DRAM to achieve better performance. It employs an enhanced persistent memory allocator to avoid performance degradation caused by expensive persistent memory allocation operations. Algorithms of operations (e.g., insertion) that require a memory allocation are carefully designed so that persistent memory leaks are prevented. HART maintains a lock on each ART to enable concurrent writes on different ARTs. We implemented HART, WOART (Write Optimal Adaptive Radix Tree) [7], ART+CoW (Adaptive Radix Tree with Copy-on-Write) [7], and FPTree [8]. After using three workloads (i.e., Dictionary, Sequential, Random) to evaluate them, we found that HART outperforms WOART [7] and FPtree [8] in most cases. In the best scenarios, HART outperforms WOART [7], ART+CoW [7], and FPTree [8] by 4.1x/3.3x/2.4x/2.3x, 5.4x/4.4x/2.4x/2.3x, and 4.0x/7.1x/4.6x/5.4x in insertion/search/update/deletion.

The rest of this paper is organized as follows. Section II explains the challenges for developing a persistent tree and related work. Section III presents the design and implementation details of HART. Section IV evaluates the performance of HART. Section V concludes this research.

II. Background

In this section, we first briefly introduce ART. Next, we explain the challenges of developing an efficient persistent tree. Finally, we summarize existing persistent trees.

A. Adaptive Radix Tree (ART)

A radix tree exhibits several features desirable for PM (see Section I). However, it has a poor utilization of memory and cache space when the keys are sparsely distributed [7]. To solve this issue, Leis *et al.* proposed ART (Adaptive Radix Tree), which adaptively chooses compact and efficient data structures for internal nodes [10]. Since the number of entries in a node could vary greatly, instead of enforcing all nodes same size, ART employs four node types (i.e., NODE4, NODE16, NODE48, and NODE256) to accommodate nodes with different numbers of entries [10]. For example, NODE4 is the smallest node type, which uses two 4-element arrays to store up to 4 keys and 4 child pointers, respectively. Along the same line, NODE16 is used for storing between 5 and 16 keys as well as child pointers. Path compression and lazy expansion further allow ART to efficiently index long keys by collapsing nodes, and thus, lowering the tree height [10]. More details of ART can be found in [10].

B. Challenges of Developing A Persistent Tree

a) Data consistency guarantee: Data consistency is an essential requirement for a persistent tree as it guarantees that all data stored in the tree can survive a system failure like a process crash or a power outage. However, current processors only support a 8-byte atomic memory write [6]. Updating a piece of data with a larger size requires some mechanisms like logging or CoW [6]. To implement these mechanisms, a certain write order has to be enforced. For example, to ensure a pointer to a valid content, updating a pointer to a leaf node has to be done **after** the leaf node itself is modified. Unfortunately, memory writes could be reordered by CPU cache or memory controller for performance purpose. Data consistency demands a careful design of a persistent tree.

b) High data consistency overhead: An existing solution to ensure ordered persistent memory writes is to employ a sequence of {MFENCE, CLFLUSH, MFENCE} instructions [6], [7]. The two instructions are supported by Intel processors. However, a study discovered that CLFLUSH significantly increases the number of cache misses, and thus, degrades performance substantially [6]. How to ensure consistency while minimizing its cost remains a difficult task.

c) Persistent memory leaks: Different from a memory leak on DRAM, a persistent memory leak is much more severe as the leaked memory cannot be reclaimed by restarting a process or system. For example, assume a system crash occurs right after a persistent leaf node of a tree is allocated. The persistent memory allocator will mark the memory space as used. However, the tree structure loses track of the leaf node. Thus, the persistent memory space taken by the leaf node can never be reclaimed, which results in a persistent memory leak. To prevent it, a persistent tree has to maintain persistent pointers to keep track of each persistent memory allocation.

C. Existing Persistent Trees

While HART and FPTree [8] target a DRAM-PM hybrid memory system, all rest trees aim at a pure PM memory system. CDDS B-Tree is a multi-version B-tree for PM [3]. The side effect of versioning is that it could generate many dead entries and dead notes. NV-Tree utilizes two new strategies: an append-only update strategy and a selective consistency strategy. Unfortunately, each split of the parent of the leaf node leads to the reconstruction of the entire internal nodes, which incurs a high overhead. A write atomic B$^+$-tree called wB$^+$-Tree requires expensive logging or CoW for a node split [9]. FPTree employs fingerprints, which are one-byte

hashes of in-leaf keys [8]. By scanning a fingerprint first, the number of in-leaf probed keys is limited to one, which leads to a significant performance improvement [8]. The three persistent trees WORT (Write Optimal Radix Tree), WOART, and ART+CoW (an ART using CoW to ensure its consistency) are all based on a radix tree [7]. Experimental results from [7] show that they outperform NV-Tree [6], wB$^+$-Tree [9], and FPTree [8]. Among the three trees, WOART performs the best in most cases [7]. Since WOART is also a variant of ART, we select it as a competitor of HART. We compare HART with FPTree as both target a DRAM-PM hybrid memory system. None of the existing persistent trees is open-sourced.

III. HART Design

In this section, we first present our design principles. Next, we elaborate the algorithms of various operations.

A. Design Principles

1. Hash-assisted ARTs. After analyzing the characteristics of hash table and ART, we find that integrating a hash table into ARTs can generate a new indexing data structure that can enjoy their complementary merits while avoiding their respective shortcomings. On the one hand, the use of a hash table can reduce the time complexity of operations on an ART. On the other hand, using an ART can escape the three limitations of a hash table (see Section I). HART limits the length of each key in the hash table so that the performance degradation caused by hash collisions is effectively reduced.

Assume that the length of each key is k bytes. In an ART, the time complexity of an insertion/search operation is $O(k)$ when there is no key compression. In a HART, the first k_h bytes of a key are used as a hash key and the rest $k-k_h$ bytes ae used as an ART key. Since k_h is a predefined parameter in HART, it will not grow when the key length k increases. Thus, the hash collision rate is always in a low range and the time complexity of an insertion/search operation in the hash table is close to $O(1)$. Therefore, the overall time complexity of an insertion/search operation in a HART is $k - k_h + 1$, which is less than that of an ART when $k_h > 1$. However, hash table shows poorer insertion performance than ART. By distributing first k_h bytes of each key in the hash table, frequently inserting new keys in the hash table can be avoided. In fact, for a sequential workload, the hash table only needs to insert a new key periodically because the first k_h bytes of a key are expected to be identical for a period of time. For a random workload, the frequency of inserting a new key to the hash table decreases after more and more keys have been inserted. Also, no update is needed as the value in a hash node is the address to an ART, which remains the same unless a reconstruction operation happens.

Fig. 1 shows the structure of HART. A key $AABF$ is split into AA (i.e., a hash key) and BF (i.e., an ART key). HART first uses the hash key to locate the AA node in the hash table, which contains a pointer to its corresponding ART (i.e., ART 1 shown in Fig. 1). All keys in ART 1 share the same prefix AA. After ART 1 is located, the ART key BF is used to find the leaf node that contains AABF. The complete key AABF is stored in the leaf node for the purpose of failure recovery.

2. Selective consistency/persistence. Similar to FPTree [8], HART also adopts a selective consistency/persistence strategy. As shown in Fig. 1, HART keeps the leaf nodes on PM while leaves all internal nodes and the hash table on DRAM. In fact, an ART does not need to store a key in a leaf node because the path to a leaf node represents the key of that leaf. Still, HART persistently stores complete keys in leaf nodes so that all critical information is durable. This selective consistency/persistence strategy offers two benefits: First, the performance (especially, write performance) of most existing PM technologies is still much lower than that of DRAM. For example, while the write latency of PCM is normally above 150 *ns*, DRAM write latency is only 15 *ns* [6]. Thus, storing internal nodes in DRAM can improve the overall performance, especially for insertion operations. Second, since HART can rebuild reconstructable data (i.e., the hash table and internal nodes) onto DRAM based on the critical data stored on PM (i.e., leaf nodes), it only needs to maintain the consistency of critical data on PM. Thus, a noticeable data consistency maintenance overhead can be saved.

3. Concurrent access. A finer granularity of locking can increase the concurrency of an indexing data structure. However, it also raises lock maintenance overhead. To make a good trade-off between concurrency and overhead, HART adopts a locking mechanism that maintains a read/write lock on each ART shown in Fig. 1. Thus, the maximal number of concurrent writes allowed by a HART is equal to its number of ARTs.

4. An enhanced persistent memory allocator. Existing persistent memory allocators exhibit poor performance when allocating numerous small objects [11], [12]. We develop a new persistent memory allocator called EPallocator (enhanced persistent memory allocator) on top of an existing PM allocator. Each time EPallocator is called, instead of allocating only one item (e.g., a leaf node or a value object), it allocates a memory chunk with multiple items so that the average cost of a single allocation is reduced. Unlike a B$^+$-tree, leaf nodes of an ART are not linked together, and thus, it cannot be recovered after a system crash. An intuitive solution is to

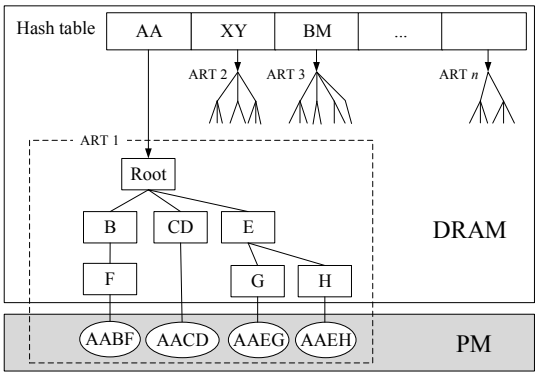


Fig. 1: The structure of HART.

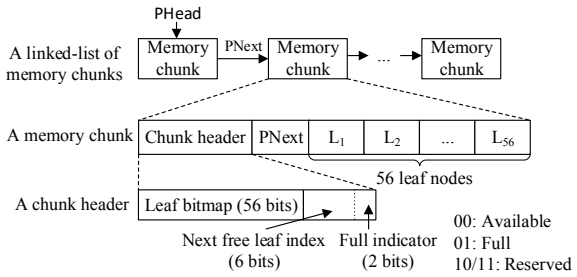


Fig. 2: A memory chunk of leaf nodes.

add a next pointer to each leaf node so that all leaf nodes are traversable [8]. However, we found that the space and performance overhead caused by maintaining a next pointer for each leaf node on PM is high. To solve this issue, EPallocator groups memory chunks in a singly linked-list so that one persistent next pointer is needed for each memory chunk rather than for each leaf node.

Fig. 2 shows a memory chunk of leaf nodes. Within each memory chunk, in addition to the 56 leaf nodes, there are two more fields: a 8-byte chunk header and a 8-byte pointer to the next memory chunk (i.e., $PNext$). The first 7 bytes of a chunk header serve as a leaf bitmap to indicate the status of leaf nodes. If a bit is set to 1, the corresponding leaf node is *used*. Otherwise, it is *free*. The last byte is split into two parts: the first 6 bits are used as an index to the next free leaf node in the 56-element leaf node array and the last 2 bits are employed as an indicator, which shows whether or not a free leaf node exists in a memory chunk (see Fig. 2). If "full indicator" is "00", the memory chunk has at least one free leaf node. If it is "01", there is no free leaf node as the memory chunk is already full. The rest two values (i.e., "10" and "11") are reserved.

5. Variable-size values support. HART also supports variable-size values. Fig. 3 illustrates the layout of a leaf node and a memory chunk of 56 value objects. Instead of keeping the value of a key in a leaf node, HART stores a 8-byte pointer (i.e., *p_value*) to the value in the leaf node. Although this design choice incurs an extra cost as EPallocator has to allocate and free PM space for the out-of-leaf value objects, it supports variable-size values that many applications require. As shown in Fig. 3, EPallocator manages PM space for value objects in the same way as what it does for leaf nodes. To support variable-size values, it maintains multiple singly linked-lists of memory chunks of value objects so that all value objects in one linked-list have the same size. The structure of a value object memory chunk is the same as that of a leaf node memory chunk as shown in Fig. 2. For simplicity, HART currently only supports two sizes of value objects: 8-byte values and 16-byte values. However, it can be easily extended to support more sizes of values by implementing more singly linked-lists of value object memory chunks. Algorithm 2 shows how EPallocator allocates PM for a leaf node or a value object.

For a radix tree and its variants, key length directly determines the height of a tree if a compression technique is not utilized. Even with a compression technique, for a radix tree with a random or dense key distribution key length is still a leading factor that determines its height. Although HART supports variable-size keys, it sets a limit on the maximal key length. The maximal key length supported by HART is 24 bytes, which could generate 2^{192} distinct keys.

6. Memory leak prevention. A persistent memory leak is more severe than a volatile memory leak as the leaked memory cannot be reclaimed through a system reboot. EPallocator can prevent persistent memory leaks. The memory chunk data structure shown in Fig. 2 is stored on PM so that the addresses of items are durable. The bitmap of a memory chunk is used to maintain the status of each item (i.e., either a leaf node or a value object). HART only sets the corresponding bit in the bitmap after an item is successfully inserted to it. If a system crash happens after a leaf node is allocated but before it is inserted to the tree, the space of the allocated leaf node can be reused as its bit in the leaf bitmap indicates that its status is still *free*. For value update and memory chunk recycling, HART utilizes a log mechanism to ensure consistency (see Algorithm 3 and Algorithm 6).

B. Algorithms of Operations

In this section, we present the algorithms of the four basic operations (i.e., insertion, update, search, and deletion) plus recovery. Similar to existing persistent trees [6]–[8], we use a sequence of {MFENCE, CLFLUSH, MFENCE} as a persistent flush instruction, which is called $persistent()$. Also, we explain $EPMalloc()$ and $EPRecycle()$, which are two components of EPallocator. They are used for persistent memory allocation and freeing, respectively.

1. *Insertion*. Algorithm 1 presents the pseudo code of an insertion operation. The first step of an insertion is to find an ART based on a hash key, which is the first several bytes of a complete key *K* (line 1-2). If the ART is not found, a new ART is initialized and then linked to the corresponding hash node (line 3-5). Next, a search operation on the ART is performed (line 6). If a leaf node with the same key *K* exists, the $Update()$ function is called to update its value (line 7-8). The algorithm of $Update()$ is shown in Algorithm 3. If the leaf node is not found, a new leaf node and a space for its value (i.e., a value object) are then allocated through $EPMalloc()$ (line 10-11). The *EPMalloc()* function is shown in Algorithm 2. It can allocate two types of objects: LEAF (i.e., a leaf node)

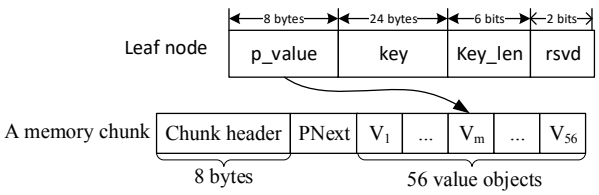


Fig. 3: Leaf node layout and a memory chunk of value objects.

and VALUE (i.e., a value object). After the value and the pointer to the value become persistent, the corresponding bit in the value bitmap (see Fig. 3, hereafter, the value bit) is set (line 12-14). Next, the key and its length are updated (line 15-16). The new leaf node is then inserted into HART in a way similar to a conventional ART insertion operation (line 17), which might lead to multiple internal node creations or expansions. Finally, the corresponding bit in the bitmap for the leaf node (hereafter, the leaf bit) is set. If a failure happens between line 14 and line 18, the value bit has been set but the leaf bit has not, which implies an exception. When EPallocator tries to allocate the same leaf node next time, it will detect the exception. As a result, it frees the PM space for the value object (line 14-15 of Algorithm 2).

EPallocator allocates PM space for both a leaf node and its value. As shown in Algorithm 2, *EPMalloc()* searches through a linked-list of leaf/value memory chunks to find a memory chunk that contains a free object (line 1-7). If no such memory chunk is found, a new memory chunk is allocated and then added to the linked-list. Next, a free object is obtained from the newly allocated memory chunk (line 8-11). If the allocation is for a leaf node, before *EPMalloc()* returns a free leaf object it has to check whether there is a value linked to the leaf object due to a prior incomplete insertion or deletion operation. If there is a value linked to the leaf object, *EPMalloc()* simply resets the value bit to make the value object available for a future allocation (line 12-16).

Algorithm 1 Insertion(Key K, Value V, HART HT)

```
1:  HashKey, ARTKey = SplitKey(K)
2:  T = HashFind(HashKey, HT)
3:  if !T then
4:      T = NewART()
5:      HashInsert(HashKey, T)
6:  leaf = SearchNode(ARTKey, HT)
7:  if leaf then
8:      Update(ARTKey, V, leaf)
9:  else
10:     leaf = EPMalloc(LEAF)
11:     value = EPMalloc(VALUE)
12:     value = V; persistent(value)
13:     leaf.p_value = &value; persistent(leaf.p_value)
14:     Set and persistent() the corresponding value bit
15:     leaf.key = K; persistent(leaf.key);
16:     leaf.key_len = len(K); persistent(leaf.key_len)
17:     Insert2Tree(T, leaf)
18:     Set and persistent() the corresponding leaf bit
```

2. *Update*. HART employs an out-of-place update mechanism. To ensure consistency and prevent memory leaks, it requires an update log, which contains three persistent pointers: *PLeaf*, *POldV* and *PNewV*. First, the address of the leaf node to be updated is recorded in *PLeaf* and then the address of the leaf node's old value is stored in *POldV* (line 2-3 in Algorithm 3). Next, the new value is written into a newly allocated space

Algorithm 2 EPMalloc(Type type)

```
1:  current_chunk = GetChunkHead(type)
2:  while current_chunk != NULL do
3:      object = GetFreeObject(current_chunk)
4:      if object != NULL then
5:          break
6:      else
7:          current_chunk = current_chunk.PNext
8:  if object == NULL then
9:      new_chunk = AllocMemChunk(type)
10:     Insert2ChunkList(GetChunkHead(type), new_chunk)
11:     object = GetFreeObject(new_chunk)
12: if type == LEAF then
13:     if object.p_value && GetBitmap(object.p_value) then
14:         Reset and persistent() the value bit
15:         EPRecycle(MemChunkOf(object.p_value))
16:         object.p_value = NULL
17: return object
```

(line 4-5). Further, *PNewV* is set to the address of the new value, after which HART sets the value bit for the new value and then updates the value pointer in the leaf node (line 7-8). The value bit for the old value is then reset and *EPRecycle()* (see Algorithm 6) checks whether the memory chunk that the old value belongs to can be freed (line 9-10). Finally, the update log is reclaimed (Line 11).

After a system crash happens, a failure recovery process checks the update log. If only *PLeaf* is valid, it simply resets the log. If both *PLeaf* and *POldV* are valid but *PNewV* is invalid, the crash happened between line 3 and line 6. In this case, since the old value is still valid and the space for the new value can be reused as its value bit has not been set, the failure recovery process simply resets the update log. If all three pointers are valid, the system crash must happen somewhere between line 7 and line 10. In this case, the recovery process resumes the update process from line 7.

Algorithm 3 Update(Key K, Value V, Leaf_Node L)

```
1:  ulog = GetMicroLog(UPDATE)
2:  ulog.PLeaf = &L; persistent(ulog.PLeaf)
3:  ulog.POldV = L.p_value; persistent(ulog.POldV)
4:  new_value = EPMalloc(VALUE)
5:  new_value = V; persistent(new_value)
6:  ulog.PNewV = &new_value; persistent(ulog.PNewV)
7:  Set the bit in the bitmap for new value
8:  L.p_value = &new_value; persistent(L.p_value)
9:  Reset the bit in the bitmap for the old value
10: EPRecycle(MemChunkOf(L.p_value))
11: LogReclaim(ulog)
```

3. *Search*. First step of the search algorithm (Algorithm 4) is to find the corresponding ART by the searching the hash table (line 1-2). After an ART is found, the search algorithm is similar to an ART search algorithm. The only difference is that

after a leaf node is found, HART will check the corresponding bitmap to make sure that it is a valid leaf node (line 9-10).

Algorithm 4 Search(Key K, HATRT HT)

```
1: HashKey, ARTKey = SplitKey(K)
2: T = HashFind(HashKey, HT)
3: if !T then
4:     return NOT_FOUND
5: leaf = SearchNode(ARTKey, T)
6: if !leaf then
7:     return NOT_FOUND
8: else
9:     if The corresponding leaf bit is set  then
10:        return leaf.p_value
11:    else
12:        return NOT_FOUND
```

4. ***Deletion***. A deletion operation is illustrated in Algorithm 5. First, the corresponding ART is found through the hash table (line 2). Second, HART uses a conventional ART search function to locate the leaf node to be deleted (line 5). After the leaf node is found, it is first removed from the tree (line 9). Third, HART resets the leaf bit and the value bit (line 11-12). Next, *EPRecycle()* is called to check whether the corresponding memory chunks can be reclaimed (line 13-14). Finally, HART will free the ART if it becomes empty after the leaf is successfully deleted (line 15-16).

Algorithm 5 Deletion(Key K, HART HT)

```
1: HashKey, ARTKey = SplitKey(K)
2: T = HashFind(HashKey, HT)
3: if !T then
4:     return NOT_FOUND
5: leaf = SearchNode(ARTKey, T)
6: if !leaf then
7:     return NOT_FOUND
8: else
9:     DeleteFromTree(T, leaf)
10:    value = leaf.p_value
11:    Reset and persistent()the leaf bit
12:    Reset and persistent() the value bit
13:    EPRecycle(MemChunkOf(value))
14:    EPRecycle(MemChunkOf(leaf))
15:    if is_empty(T) then
16:        free(T)
```

Each time *EPRecycle()* is called an object is deleted from the tree. Algorithm 6 shows its procedure. Before recycling a memory chunk, *EPRecycle()* has to make sure that there is no used object in it (line 1-2 in Algorithm 6). To remove the memory chunk from the memory chunk linked-list, a persistent recycle log is needed to ensure the consistency of the linked-list. The recycle log contains two persistent pointers: *PPrev*, and *PCurrent*. First, *PCurrent* is set to point to the memory chunk to be deleted (line 4). If *PCurrent* is not *PHead*, which is a pointer to the head of a linked-list (see Fig. 2), then the address of *PPrev* is also stored in the log (line 8-9). Next, the next pointer of previous memory chunk is updated (line 10), after which the memory chunk is freed and the log is reset (line 11-12). If a failure happens before the log is reclaimed, a failure recovery process will check the log. If both *PPrev* and *PCurrent* are set, the deletion can be resumed from line 10. If only *PCurrent* is valid, the failure recovery process has to compare it with *PHead*. If *PCurrent* = *PHead*, the deletion can be resumed from line 6. If *PCurrent.PNext* = *PHead*, the deletion can be resumed from line 11.

Algorithm 6 EPRecycle(Mem_Chunk mem_chunk)

```
1: if mem_chunk has a used object then
2:     return
3: rlog = GetMicroLog(RECYCLE)
4: rlog.PCurrent = &mem_chunk; persistent(rlog.PCurrent)
5: if rlog.PCurrent == PHead then
6:     PHead = mem_chunk.PNext; Persist(PHead);
7: else
8:     pre = GetPrev(mem_chunk)
9:     rlog.PPrev = &pre_chunk; persistent(rlog.PPrev)
10:    pre.PNext = mem_chunk.PNext; persistent(pre.PNext)
11: pfree(mem_chunk)
12: LogReclaim(rlog)
```

5. ***Recovery***. Since HART only stores leaf nodes on PM, all internal nodes need to be recovered after a system crash or a system reboot. Recovering a HART is much faster than building a new HART from scratch because the leaf nodes and values are already on PM before a recovery process starts. As shown in Algorithm 7, the recovery process first initialize a new HART (see line 1), which allocates space for the hash table. Then the recovery process traverses all memory chunks through the memory chunk list to recover a HART. For each memory chunk, only the leaf nodes whose leaf bits are in a "set" status will be inserted to the tree. Function *Insert2HART()* in line 6 is similar to Algorithm 1.

Algorithm 7 Recovery(HART HT)

```
1: InitializeHART(HT)
2: current_chunk = GetChunkHeade(LEAF)
3: while current_chunk do
4:     for i = 0; i < NUM_OBJECTS_PER_CHUNK; i++ do
5:         if The corresponding bit in the bitmap is set then
6:             Insert2HART(HT, &(current_chunk.leaf_array[i]))
```

IV. Evaluation

In this section, we first introduce experimental setup. Next, we discuss our experimental results of the four trees.

A. Experimental Setup

We implemented HART and three existing persistent trees (i.e., WOART [7], FPTree [8], and ART+CoW [7]) in C

language. The source code of HART is available on GitHub (https://github.com/CASL-SDSU/HART). While the implemented FPTree is based on an open-source implementation of B$^+$-Tree [13], the rest three trees were implemented based on an open-source implementation of ART [14]. We did not compare HART with HiKV [5] because some critical implementation details (e.g., the hash function used) were not disclosed in [5].

Existing PM research studies [1], [4], [7], [12] have to leverage a PM emulator due to the lack of real PM hardware. Intel PMEP (Persistent Memory Emulator Platform) [1], [15] and Quartz [16] are two commonly used PM emulators. However, Intel PMEP was no longer public available at the time of this research. As for Quartz, we found that when it was running in the DRAM-PM hybrid mode it needs to frequently call $numa\_alloc\_onnode()$, which emulates an allocation of a piece of PM by executing a DRAM allocation on a remote node. Unfortunately, we observed that when the number of invocations of $numa\_alloc\_onnode()$ was large enough (e.g., more than 10 million) the experimental results provided by Quartz became meaningless as they were greatly distorted. We discovered that this is because the software latencies of Quartz caused by $numa\_alloc\_onnode()$ became dominant such that real PM latencies were concealed. Therefore, we employed two methods to emulate PM write latencies and PM read latencies, respectively. To emulate PM write latency, similar to current work [5], [16], we added the write latency difference between PM and DRAM to each invocation of $persistent()$, which flushes a piece of data from CPU cache to memory. To emulate PM read latency, we have to consider the effect of CPU cache hits. We calculated the extra latency caused by the read latency difference between DRAM and PM by using the following two equations proposed by [15], [16]:

$$\delta_{stall\_cycles} = S * (L_{PM} - L_{DRAM})/L_{DRAM}, \quad (1)$$

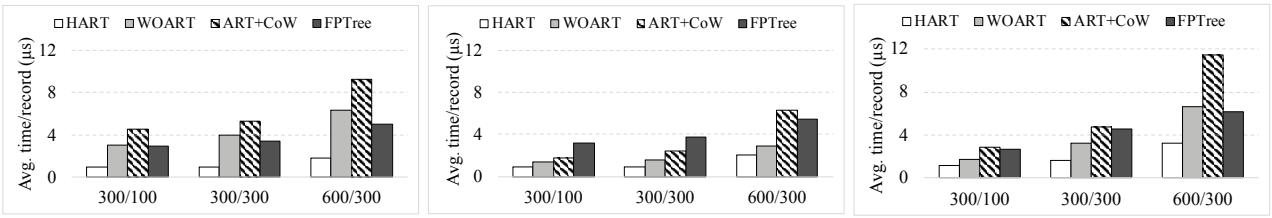
$$\delta_{r\_latency} = \delta_{stall\_cycles}/CPU\_frequency, \quad (2)$$

where δ_{stall_cycles} is the extra number of CPU stall cycles caused by the read latency difference between DRAM and PM, S is the total number of CPU cycles that a processor has stalled due to serving all LOAD memory requests on a remote node during an experiment, L_{DRAM} is the latency of DRAM, L_{PM} is the desired PM latency, and $\delta_{r\_latency}$ is the extra latency caused by the read latency difference between DRAM and PM. Fortunately, Quartz [16] provides the statistics of S. Thus, we run it to conduct an experiment so that we can obtain S to calculate additional read latency off-line. Authors of [17] also adopted this PM read latency off-line adding method.

We conducted all experiments on a Mercury RM102 1U Rackmount Server running Ubuntu 16.04 with Kernel 4.4.0. It has two sockets each equipped with one Intel Xeon E5-2640 v3 2.6 GHz processor. Each processor has 8 cores, a shared 20 MB L3 cache, and 32 GB DRAM. The server is organized into 2 NUMA nodes (i.e., node 0 and node 1) with each having a processor and 32 GB local memory. All single-threaded experiments were running on node 0 (i.e., the local node) whose local DRAM is treated as DRAM. The DRAM in the remote node (i.e., node 1) is taken as PM. A PM allocation is emulated by calling $numa\_alloc\_onnode()$, which in fact allocates a DRAM space from the remote node. We measured the local DRAM latency and remote DRAM latency. They are about 100 ns and 150 ns, respectively. To avoid the problem associated with $numa\_alloc\_onnode()$, we run each experiment in two rounds. In the fist round, we run it in a pure DRAM environment (i.e., each invocation of $numa\_alloc\_onnode()$ was replaced by a calling to $malloc()$) to obtain a baseline execution time of the experiment. In the second round, we run the experiment again in a DRAM-PM hybrid environment (i.e., enabling $numa\_alloc\_onnode()$) so that we can obtain the total number of CPU stall cycles S. The write latency difference between DRAM and PM has been added to each invocation of $persistent()$ in the first round because the execution time measured in the second round cannot be used. Next, we employed the two equations to obtain the extra latency caused by the read latency difference between DRAM and PM, which was then added to the baseline execution time. Finally, we obtained the execution time of the experiment on an emulated DRAM-PM memory system.

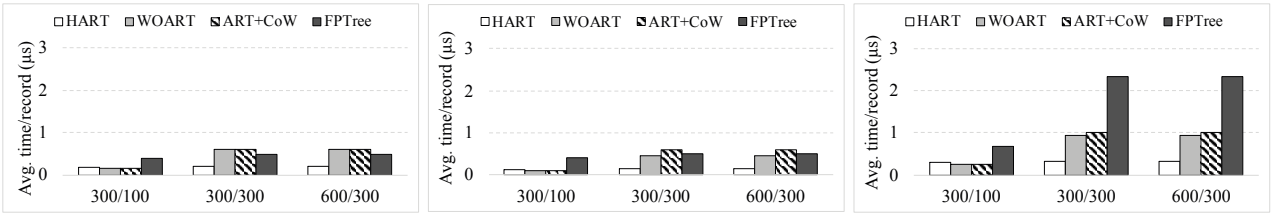
We compiled all four tree implementations using GCC 5.4.0. For HART, the hash key length is set to 2 in our experiments. We used three workloads, namely, Dictionary [19], Sequential, and Random. Dictionary is a collection of 466,544 different English words [19]. Sequential and Random are two synthetic traces generated by ourselves. Sequential contains sequential strings, whereas Random includes random strings with variable sizes from 5 to 16 bytes. For Sequential and Random, each character in a key is chosen from the 52 alphabetic characters (i.e., A to Z and a to z) and 10 Arabic numerals (i.e., 0 to 9). A typical range of PM latencies is from several ns to 1,000 ns [20]. Since we use DRAM to emulate PM, we can only imitate PM latency larger than or equal to 100 ns, which is the measured DRAM latency on the server that we used. Three PM write/read latency configurations were employed in our experiments: 300 *ns*/100 *ns*, 300 *ns*/300 *ns*, and 600 *ns*/300 *ns*. Similar PM write/read latency settings have been used in existing PM research such as FPTree [8] and NOVA [4]. For simplicity, hereafter the three latency configurations are called 300/100, 300/300, and 600/300, respectively. For 300/100, the default number of records on Sequential and Random is set to 100 million. For the other two PM latency configurations, we set the default number of records to 1 million for Sequential and Random to prevent memory allocation failures caused by $numa\_alloc\_onnode()$. Note that in 300/100 we only need to run each experiment in the first round because in this configuration the read latency of PM is equal to that of DRAM.

Thus, the problem of $numa\_alloc\_onnode()$ does not exist as a call for the function is not needed. That is why we can test up to 100 million records in 300/100. We measured the performance of the four persistent trees in terms of average time per operation for four basic operations: insertion, search,



(a) Dictionary (b) Sequential (c) Random

Fig. 4: Insertion performance comparisons.



(a) Dictionary (b) Sequential (c) Random

Fig. 5: Search performance comparisons.

update, and deletion. We also tested range query, memory consumption, recovery, and scalability of HART.

B. Performance of Four Basic Operations

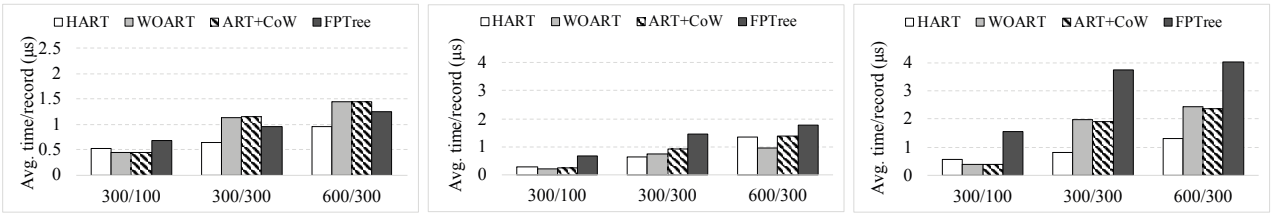
Insertion: Fig. 4 shows the four persistent trees' average times of inserting one record under the three workloads and three PM write/read latency configurations. Results from Fig. 4 demonstrate that HART consistently outperforms all its competitors. Compared with WOART, in the best case HART is 4.0x faster under Dictionary in 300/300 (Fig. 4a). In the worse case, HART is 1.4x faster under Sequential in 600/300 (Fig. 4b). This is because HART stores internal nodes in DRAM while WOART keeps them on PM. Therefore, HART does not need to maintain the consistency for internal nodes, and thus, that overhead can be saved. Compared with FPTree, in the best case HART is 4.0x faster under Sequential in 300/300. In the worst case, HART is still 1.9x faster under Random in 600/300 (Fig. 4c). HART performs much better than FPTree because it does not need to search on unsorted leaf nodes. In short, the performance differences between DRAM and PM as well as the reductions of $persistent()$ calls substantially boost the insertion performance of HART. Fig. 4 shows that in most cases ART+CoW performs the worst. The main reason is that its CoW overhead is very high.

Search: Fig. 5 illustrates the search performance of the four persistent trees. HART shows better search performance under 300/300 and 600/300 across all three workloads. We notice that when PM read latency is equal to that of DRAM (i.e., 300/100), WOART achieves better performance than HART. The reason is that for a read-only operation, HART consumes more memory, which results in a lower cache hit rate. We also find that FPTree is faster than WOART under Dictionary in 300/300 and 600/300. The reason is that Dictionary has a relatively small number of records, and thus, the height of FPTree is also small. However, the height of WOART is independent of the number of records. FPTree performs much better under Sequential than under Random because its performance is closely related to the cache hit rate as each leaf node contains multiple records.

Update: In our implementations, we used a similar update mechanism for HART, WOART, and ART+CoW: since all three support variable-size values, only the pointer to a value is stored in each leaf node. During an update, a new PM space is allocated for the new value. A pointer to that new value is updated as the last step to ensure consistency. Although the update functions are similar in the three ART-based trees, we can see from Fig. 6 that HART still outperforms WOART and ART+CoW in most cases. The performance improvements of HART come from its capability of quickly searching an existing leaf node. For the same reason, HART outperforms FPTree in all cases.

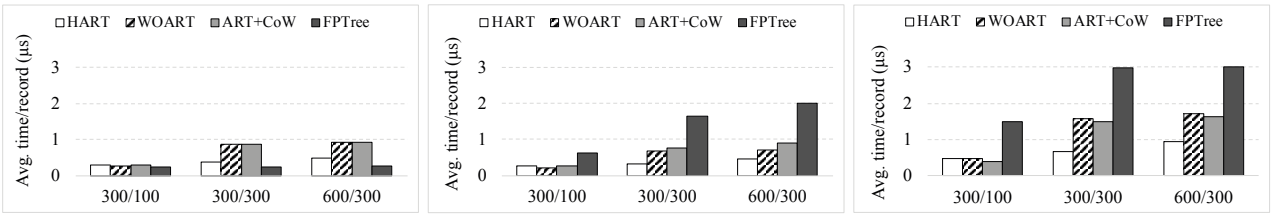
Deletion: A deletion operation requires a search operation to find the leaf to be deleted. Fig. 7 shows that under Dictionary FPTree achieves the best performance. The reason is twofold. First, Dictionary has a relatively small amount of records (i.e., 466,544 records), which makes the height of FPTree low. Thus, the search operation can be finished quickly. Second, FPTree does not perform tree re-balancing on leaf nodes, which accelerates a deletion. Under Random and Sequential, since FPTree no longer has the tree height advantage as the number of records in each workload is at least one million, it shows the worst performance. In general, HART exhibits its strength when the PM latency is set higher than that of DRAM for a larger data set.

To understand the impact of the number of records on the performance of the four basic operations, we evaluate the four



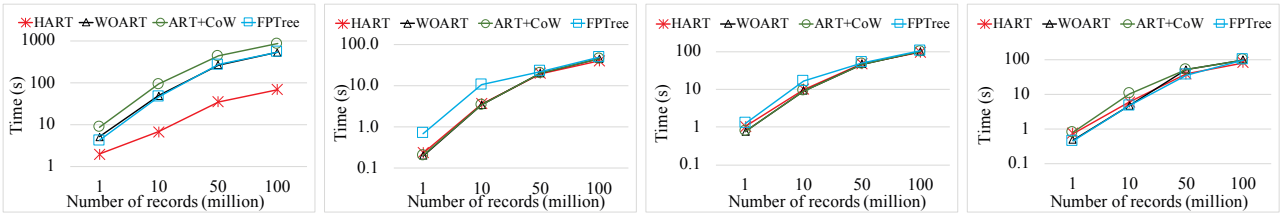
(a) Dictionary (b) Sequential (c) Random

Fig. 6: Update performance comparisons.



(a) Dictionary (b) Sequential (c) Random

Fig. 7: Deletion performance comparisons.



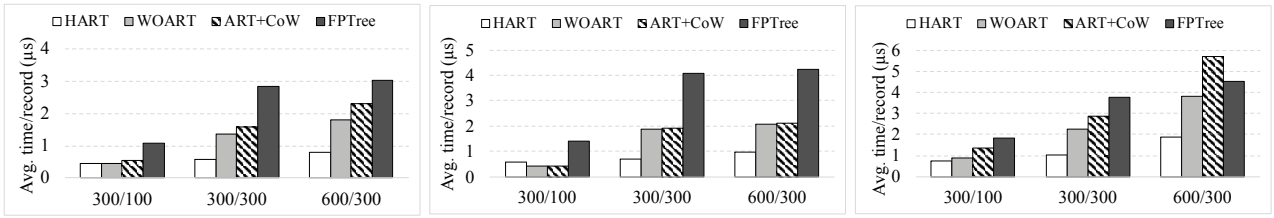
(a) Insertion (b) Search (c) Update (d) Deletion

Fig. 8: Impact of the number of records on the four basic operations.

trees in Fig. 8 when the number of records increases from 1 million to 100 million under Random in 300/100. Fig. 8a shows that HART exhibits a much better scalability in terms of insertion. In search, FPTree shows the worst performance because it has to conduct key comparisons (see Fig. 8b). The three ART-based trees deliver very similar performance in search and update (see Fig. 8b and Fig. 8c). In deletion, there are little differences among HART, WOART, and FPTree (see Fig. 8d). Note that the latency differences between DRAM and PM are not substantial in 300/100. That is why HART shows a performance similar to that of WOART in three operations. However, after the latency differences enlarge (i.e., 300/300 or 600/300) HART offers a much better performance than WOART and FPTree (see Fig. 4 - Fig. 7).

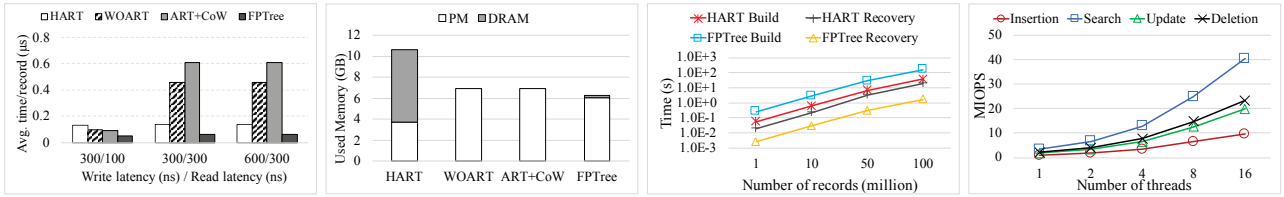
C. Performance of Mixed Workloads

To understand the performance of HART under realistic benchmarks, we measured the performance of the four trees using three typical cloud database workloads generated by YCSB (Yahoo! Cloud Serving Benchmark) [21]. YCSB is a standard benchmarking framework to evaluate various cloud key-value stores that provide online read/write access to data [21]. Each workload generated by YCSB represents a particular mix of read/write operations and a specific request distribution, which decides which record in a database to read or write. The three mixed workloads that were used in our experiments all employ a Uniform request distribution, which means that all records in the database are equally likely to be chosen when a read or write request arrives [21]. They cover three typical online database workload scenarios: (1) Read-Intensive: a workload with 10% insertion, 70% search, 10% update, and 10% deletion; (2) Read-Modified-Write: a workload with 50% search and 50% update; (3) Write-Intensive: a workload with 20% search, 40% insertion, and 40% update. While Read-Intensive and Write-Intensive stand for two extreme cases, Read-Modified-Write represents a read-write-balanced scenario. Experimental results from the three mixed workloads are presented in Fig. 9, which shows that HART outperforms its three competitors in almost all cases. The only exception is for the Read-Modified-Write workload under 300/100 (see Fig. 9b). WOART and ART+COW performs better than HART in that case because they exhibit a lower latency in both search (see Fig. 5) and update (see Fig. 6) operations under 300/100. Other than the 300/100 setting,



(a) Read-Intensive (b) Read-Modified-Write (c) Write-Intensive

Fig. 9: Performance comparisons under three mixed workloads.



(a) Range query (b) Memory consumption (c) Recovery (d) HART scalability

Fig. 10: Performance in range query, memory consumption, recovery, and HART scalability.

HART is 2.0x to 2.6x faster compared with WOART under all three mixed workloads. Also, it is 2.4x to 5.7x faster compared with FPTree under all three mixed workloads. The conclusion is that HART outperforms its three competitors under realistic workloads in almost all cases.

D. Performance of Range Query

We tested the range query performance by querying 100,000 records for the four persistent trees under Sequential. In fact, the range query function in the three ART-based trees are simply implemented by calling a search function for each key. For FPTree, since its leaf nodes are ordered in the linked-list, it shows the best range query performance in Fig. 10a. Compared with FPTree, the range query performance of HART is 2.6x, 2.3x, 2.3x slower under 300/100, 300/300, and 600/300, respectively. Nevertheless, compared with WOART and ART+CoW, HART still shows a much better performance when the PM read latency is set to be higher than that of DRAM (i.e., higher than 100 *ns*). In fact, the side-effect of hash on range query of HART is very limited because the main part of HART are multiple ART trees (see Fig. 1).

E. Memory Consumption

We also measured the memory consumption of the four data trees. Due to space limit, we only show results under Sequential with 100 million records in Fig. 10b. Note that WOART and ART+CoW do not use any DRAM. We found that compared with FPTree, HART consumes much more DRAM. The major reason is that each character in a key is chosen from 62 different characters (i.e., A to Z, a to z, and 0 to 9). Thus, many nodes of the NODE256 type are needed as NODE48 has an insufficient space to accommodate these keys. Also, the hash table in HART takes extra DRAM space. FPTree consumes more PM space than HART does. The reason is that the fingerprints in FPTree take a large PM space. Also, FPTree does not coalesce a leaf node with its neighbor when the number of available keys in it is less than half of its capacity.

F. Performance of Recovery

Since both WOART and ART+CoW are a pure PM tree, they have no need to recover nodes after a system failure or a normal reboot. Thus, we only evaluated the recovery times for HART and FPTree in Fig. 10c. Also, we tested their build times. The build time of HART or FPTree is the time taken to generate a new HART or FPTree by sequentially inserting a number of records. All experiments were conducted under Random with 300/100. The number of records to be rebuilt varies form 1 million to 100 million. We noticed that for both trees their recovery times are shorter than their build times. On average, HART recovery is 2.4x faster than its build. However, the recovery time of FPTree is much shorter than that of HART. The reason is that each FPTree leaf node contains multiple records while a HART leaf node only has one record. As a result, FPTree needs much less insertions than HART does, which leads to a much shorter recovery time. However, we argue that tree recovery is normally not a frequent event.

G. Multi-threaded Results

To allow concurrent accesses, HART maintains an exclusive write lock and a sharable read lock for each of its ART (e.g., ART1 shown in Fig. 1). It employs the POSIX threads library (i.e., *pthread*) to implement its feature of supporting concurrent accesses. For each operation (e.g., a read operation like a search or a write operation like an insertion), HART assigns a thread to accomplish it. HART only supports concurrent writes that target distinctive ARTs. For concurrent reads, such restriction does not exist. On each ART, HART allows

multiple read threads (e.g., search or scan) to share the read lock so that they can operate concurrently on the same ART. However, on each ART it allows only one write thread (e.g., insertion, update, deletion) to hold the exclusive write lock at any time. Besides, when a write thread is working on an ART all incoming read threads on the same ART are blocked. For an incoming write operation, HART first checks whether the read lock on its destination ART is currently free. If it is not free, the write operation is blocked. Otherwise, HART further checks whether the exclusive write lock on the write operation's destination ART is presently held by another write thread. If so, the write operation is blocked until the exclusive write lock is freed. If not, a thread (i.e., a write thread) is assigned to the write operation and then the exclusive write lock is acquired by the write thread. For an incoming read operation, HART checks whether the exclusive write lock is free. If not, the read operation is blocked. Otherwise, a thread (i.e., a read thread) is assigned to the read operation. The read thread starts to work after either acquiring the free read lock or sharing it with other ongoing read threads on the same ART.

All experimental results shown from Fig. 4 to Fig. 10c were obtained when HART was executed in a single-threaded mode. We evaluated its concurrent access performance in terms of MIOPS (million I/O operations per second) using the 300/100 latency configuration and 100 million Random records in Fig. 10d. All threads were running on a single socket, which has 8 physical cores. Using Hyper-Threading, each physical core can support 2 threads. Thus, a single socket can support up to 16 threads. Fig. 10d shows that compared with single-threaded scenarios the performance of HART on 2 threads is increased by a factor of 1.96/1.94/1.93/1.93 for insertion/search/update/deletion, respectively. Also, the performance of HART increases by a factor of 7.18/7.30/7.13/7.09 for the four operations when the number of threads increases to 8 (see Fig. 10d). In fact, when the number of threads increases from 2 to 8, the performance of HART almost increases proportionally to the number of threads. However, compared with the single-threaded cases the performance of HART with 16 threads is only increased by a factor of 10.7/11.9/11.3/10.8 for the insertion/search/update/deletion, respectively. The reason is that when using 16 threads each physical core is abstracted as two logical cores by Hyper-Threading. The performance of a logical core is lower than that of a physical core. We also found that the performance improvement of search is higher than that of other three operations. This is because concurrent read threads do not block each other. Fig. 10d demonstrates that HART scales well in concurrent situations.

V. Conclusions

In this paper, we design, implement, and evaluate a persistent indexing data structure called HART. Our comprehensive experimental results demonstrate the strength of HART. Compared with WOART, HART can not only provide much better performance in most cases but also prevent persistent memory leaks, which has not been addressed in WOART. Although both FPTree and HART target a DRAM-PM hybrid memory system, HART significantly outperforms FPTree in the four basic operations due to the inherent advantages of an ART over a B^+ tree. In terms of range query and tree recovery, however, FPTree exhibits a better performance than HART. Besides, it consumes less DRAM. We will release the source code of HART for public use in the near future.

VI. Acknowledgment

We thank Ismail Oukid for his help in FPTree implementation. We thank Bo-Wen Shen for providing us with the Mercury RM102 1U Rackmount Server.

References

[1] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *EuroSys*. ACM, 2014, p. 15.

[2] (2017) The wait is over! 3d xpoint technology. [Online]. Available: http://www.intelsalestraining.com/infographics/memory/3DXPointc.pdf

[3] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory." in *FAST*, vol. 11, 2011, pp. 61–75.

[4] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories." in *USENIX Conference on File and Storage Technologies*, 2016, pp. 323–338.

[5] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *USENIX ATC*. USENIX Association, 2017, pp. 349–362.

[6] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems." in *FAST*, vol. 15, 2015, pp. 167–181.

[7] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "Wort: Write optimal radix tree for persistent memory storage systems." in *FAST*, 2017, pp. 257–270.

[8] 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 *ICMD*. ACM, 2016, pp. 371–386.

[9] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[10] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *ICDE*. IEEE, 2013, pp. 38–49.

[11] 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 *Timely Results in Operating Systems*. ACM, 2013, p. 1.

[12] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, "Memory management techniques for large-scale persistent-main-memory systems," *VLDB*, vol. 10, no. 11, pp. 1166–1177, 2017.

[13] "bpt: B+ tree implementation," 2016. [Online]. Available: http://www.amittai.com/prose/bpt.c

[14] (2017) Adaptive radix trees implemented in c. [Online]. Available: https://github.com/armon/libart

[15] S. R. Dulloor, "Systems and applications for persistent memory," Ph.D. dissertation, Georgia Tech, 2015.

[16] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *Middleware*. ACM, 2015, pp. 37–49.

[17] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the nvram era," *VLDB*, vol. 7, no. 2, pp. 121–132, 2013.

[18] "Memcached," http://http://memcached.org/, [Online; accessed 1-Januray-2019].

[19] (2017) A text file containing 479k english words for all your dictionary. [Online]. Available: https://github.com/dwyl/english-words

[20] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.