Failure-Atomic Byte-Addressable R-tree for Persistent Memory

Soojeong Cho, Wonbae Kim, Sehyeon Oh, Changdae Kim, Kwangwon Koh, and Beomseok Nam, *Member, IEEE*

Abstract—In this article, we propose Failure-atomic Byte-addressable R-tree (FBR-tree) that leverages the byte-addressability, persistence, and high performance of persistent memory while guaranteeing the crash consistency. We carefully control the order of store and cacheline flush instructions and prevent any single store instruction from making an FBR-tree inconsistent and unrecoverable. We also develop a non-blocking lock-free range query algorithm for FBR-tree. Since FBR-tree allows read transactions to detect and ignore any transient inconsistent states, multiple read transactions can concurrently access tree nodes without using shared locks while other write transactions are making changes to them. Our performance study shows that FBR-tree successfully reduces the legacy logging overhead and the lock-free range query algorithm shows up to 2.6x higher query processing throughput than the shared lock-based crabbing concurrency protocol.

Index Terms—R-tree, persistent memory, failure-atomicity, multidimensional indexing structure

1 Introduction

Recent advances in byte-addressable persistent memories (PM) such as 3D Xpoint [19], phase-change memory [54], and STT-MRAM [16] are expected to open up new opportunities to transform main memory from volatile device to persistent storage [8], [21], [26], [43], [44], [47], [49], [55], [58]. Due to its persistency and byte-addressability, PM can be used either as slow but large main memory or as fast secondary storage via legacy block I/O interfaces [4], [6], [8], [17], [24], [32], [39], [40], [57]. To leverage the high performance, persistence, and byte-addressability of PM, various opportunities are being pursued in numerous domains, including operating systems and database systems [24], [33], [45]. However, how PM will interact with existing systems has not been thoroughly investigated, and a possible role of PM for multidimensional spatial objects is currently an open question. In this work, we study how R-tree, one of the most popular data structures for multidimensional datasets, can benefit from PM.

- *Soojeong Cho and Sehyeon Oh are with the Ulsan National Institute of Science and Technology (UNIST), Ulsan 44919, South Korea. E-mail: {cristalcho, osh829}@unist.ac.kr.*
- *Wonbae Kim is with the Ulsan National Institute of Science and Technology (UNIST), Ulsan 44919, South Korea, and also with ETRI, Daejeon, South Korea. E-mail: wbkim@unist.ac.kr.*
- *Changdae Kim is with Data Centric Computing Systems, ETRI, Daejeon, South Korea. E-mail: cdkim@etri.re.kr.*
- *Kwangwon Koh is with SW Fundamental Research, ETRI, Daejeon, South Korea. E-mail: kwangwon.koh@etri.re.kr.*
- *Beomseok Nam is with Sungkyunkwan University, Seoul 110-745, South Korea. E-mail: bnam@skku.edu.*

Manuscript received 16 Nov. 2019; revised 20 July 2020; accepted 20 Sept. 2020. Date of publication 6 Oct. 2020; date of current version 15 Oct. 2020. (Corresponding author: Beomseok Nam.)
Recommended for acceptance by K. Mohror.
Digital Object Identifier no. 10.1109/TPDS.2020.3028699

To manage data structures in PM, the properties of PM must be considered. Although we can employ traditional techniques, such as logging and shadowing when designing data structures for PM, they may duplicate unmodified portions of data structures, which incurs significant overhead due to additional memory writes and cache line flush instructions. Implementing byte-addressable data structures for PM has very different characteristics from disk-based data structures as well as in-memory data structures. First, there is no guarantee that when modified dirty cachelines will be written back to PM. Even if we do not explicitly call `clflush` instructions, dirty cache lines can be flushed by cache replacement mechanisms. Such a premature cacheline flush can make data structures in PM inconsistent, and such inconsistency becomes permanent and exposed to other processes when a system crashes. Second, when processors store data in PM, they guarantee at most 8 bytes of data to be written atomically. Most data structures consist of logical blocks of composite data types; thus, failure-atomicity at a granularity of 8 bytes is not sufficient to guarantee crash consistency. Third, multiple write operations may not occur in the same order as they are written by a process running on the CPU. Such reordering of memory writes does not harm in volatile memory because applications that make changes to volatile memory can protect the inconsistent data via locking mechanisms, and partially written data will be lost when a system crashes. However, if we store data objects on PM, such transient inconsistent data written in an arbitrary order will persist across system failures.

To address these challenges associated with a fine-grained write unit in PM, various block-based indexing structures, such as B+-trees [4], [18], [49], [56] and hash tables [38], [59], [60] have been redesigned. However, to the best of our knowledge, no previous study has attempted to design multidimensional indexing structures for byte-addressable persistent memory.

1045-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.

Multidimensional range queries are an important class of problems in database systems, computer graphics, geographic information systems, and high-performance computing [2], [7], [9], [10], [12], [22], [28], [34], [36], [37], [48], [50]. In many scientific domains, the size of data files produced by scientific applications continue to grow, and petabytes or exabytes of data will soon be common. Typically, the content of scientific data files are a collection of multidimensional arrays along with the associated spatio-temporal coordinates [2], [5], [28], [36]. To help navigating through large scientific datasets, various multidimensional indexing techniques, such as R-trees [13], [29], have been developed and used widely to allow direct access to particular datasets [11], [12], [35], [37], [42], [53]. For example, multiphysics oil reservoir simulation [27], spatial modeling of the brain [48], and disease transmission analysis [5] employ multidimensional indexes to accelerate range query processing performance. For large-scale scientific datasets, persistent memory offers the ability to bring persistent data closer to the CPU and to extend the capacity of main memory. We note that Intel's latest Optane DC Persistent Memory [19] extends the capacity of DRAM and enables very large storage class memory, i.e., 3 TBytes of memory capacity per server CPU socket.

In this work, we design and implement a variant of an R-tree for byte-addressable PM. Byte-addressable PM raises new challenges in the use of R-trees because legacy R-tree operations are based on the assumption that block I/O is failure-atomic and memory operations are volatile. However, in PM, each memory operation at the granularity of a word, e.g., 8 byte, must be failure-atomic, i.e., data must be consistent for each store instruction. Otherwise, partially updated inconsistent tree nodes can be exposed to other transactions upon system failures, or to other concurrent read transactions if lock-free search is enabled. To prevent the reordering of memory write operations and to ensure that each cacheline flush to PM does not compromise the consistency of R-tree structures, we carefully redesign R-tree algorithms to control the order of memory writes and cacheline flushes so that R-tree can tolerate *transient inconsistency* [18] caused by incomplete write transactions.

The key contributions of this work are as follows.

- We design and implement byte-addressable and failure-atomic algorithms for multidimensional R-tree optimized for PM. The insert and delete algorithms consist of a sequence of 8-byte store instructions, each of which preserves R-tree invariants.
- We present a novel *in-place rebalancing with byte-addressable metadata-only logging* algorithm for R-tree node split and merge operations, and compare it against *copy-on-write (CoW)-based rebalancing* algorithm.
- We show that fine-grained control of failure-atomic 8-byte store instructions enables the lock-free search for R-trees on PM. Non-blocking queries on R-trees significantly improve the concurrency level and transaction throughput by up to 13.5x.

The remainder of this paper is organized as follows. In Section 2, we present the challenges involved in designing a R-tree index on PM. In Section 3, we present the design and implementation of failure-atomic and byte-addressable persistent R-tree (FBR-tree). In Sections 4 and 4.3, we discuss concurrency and consistency issues associated with an FBR-tree. In Section 5, we evaluate the performance of FBR-trees. Conclusion of this paper is presented in Section 6.

2 Challenges in Design of Indexing Trees for Persistent Memory

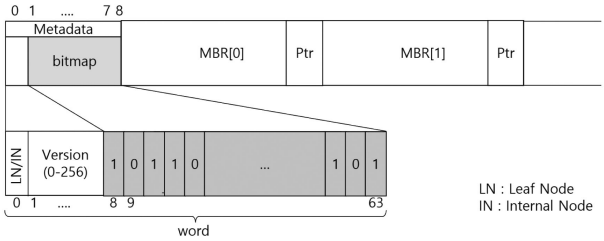
R-tree structures are similar to B-tree structures in that both structures are balanced search trees and are designed for block device storage where data items are organized in pages. Although, to the best of our knowledge, there exists no prior work that studies multidimensional indexing trees on PM, various B-tree variants for PM have been proposed to resolve the challenges of PM and benefit from its high-performance [4], [18], [41], [56].

One of the invariants of legacy B-trees is that the key-value pairs in each tree node are stored in sorted order. This invariant entails a large number of shifts and cache line flushes. In the legacy disk-based index, the overhead of a large number of shifts and cacheline flushes is negligible due to incomparably large disk I/O overhead. However, in high-performance PM, the overhead of memory barriers and cacheline flushes is known to be the dominant performance factor [4], [18], [41], [45], [49], [56]. To avoid such a large number of shifts and cacheline flushes, NV-tree [56], FP-tree [41], and wB+-tree [4] proposed to append unsorted key-value pairs into the array. The append-only update strategy has been shown to improve write performance, but at the cost of higher lookup overhead because it requires linear scanning of all unsorted keys [18].

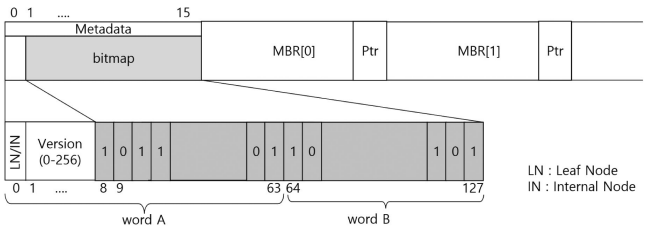
R-tree is different from B-tree in that a tree node of R-tree stores a set of Minimum Bounding Rectangles (MBR) instead of an array of sorted keys. That is, unlike B-trees, R-trees do not require the multidimensional MBRs to be geometrically sorted. Instead, R-tree requires each MBR to enclose all the MBRs of its corresponding sub-tree. I.e., one of the invariants of R-tree is the hierarchical enclosure relations of MBRs. Such an invariant is required for multidimensional queries. That is, a search for multidimensional objects has to visit all sub-trees whose associated MBRs overlap the given point. This invariant puts a unique challenge on designing a failure-atomic byte-addressable R-trees. I.e., unlike B-trees where each query traverses the tree structure in a top-down manner, R-tree queries backtrack to the previously visited parent nodes. I.e., if a query overlaps multiple MBRs in an R-tree node, all the corresponding child nodes must be visited in a depth first order. When a leaf node is reached, the spatial coordinates of objects in the leaf node are compared against the query range, and their objects are put into the result set if they lie within the search range. Since an R-tree query scans all MBRs in a visited node, the ordering of MBRs in a tree node does not affect the performance of an R-tree search.

When rebalancing a tree-structured index, recovery methods, such as logging and CoW, make the structure recoverable by writing a consistent copy elsewhere prior to updating the tree structure. However, in byte-addressable PM, such per-node logging or per-node CoW is known to be expensive and sub-optimal because it unnecessarily duplicates the entire tree node, including the unmodified

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.

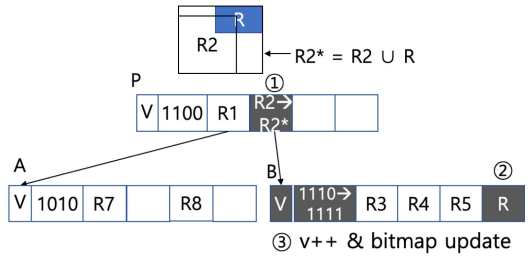


(a) Node Structure with 8 Byte Metadata

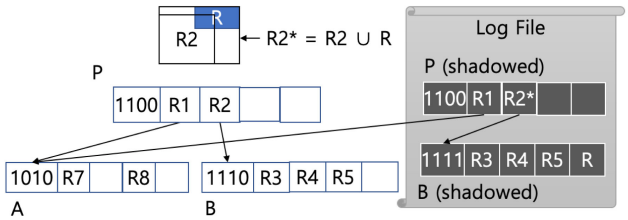


(b) Node Structure with Metadata of Unlimited Size

Fig. 1. Node structure of FBR-tree.



(a) Inserting MBR 'R' into R-tree



(b) Legacy Disk-based Logging

Fig. 2. Byte-addressable insertion in FBR-tree.

portion of it. To resolve this problem, NV-tree [56] and FP-tree [41] use *selective persistence* that keeps leaf nodes in PM but internal nodes in volatile DRAM. This is because internal tree nodes can be reconstructed from scratch when a system restarts. Although the selective persistence makes logging unnecessary, it requires reconstruction of whole tree structures upon any system fault. In that regard, NV-tree and FP-tree are not persistent indexes in strict sense.

3 Design and Implementation of FBR-tree for PM

In this section, we present the details of the tree node structure and tree operations of the proposed Failure-atomic and Byte-addressable R-tree optimized for PM with reduced consistency cost.

3.1 Node Structure

Fig. 1 shows the structure of FBR-tree nodes. If the metadata size is bounded by an 8-byte word, as shown in Fig. 1a, we can update the metadata in a failure-atomic manner with a single 8-byte store instruction. Each bit in the bitmap indicates whether its corresponding MBR and pointer are valid. If the kth bit is 0, a pair of `MBR[k]` and its pointer in the node is a free space. Otherwise, the MBR and the pointer are valid and consistent. Therefore, the maximum number of MBRs we can store in a tree node with 8-byte metadata is limited to 55. Note that we use one bit to store the type of tree node (leaf (LN) or internal node (IN)) and one byte to store the version number, which is necessary to implement a lock-free search algorithm, which we will describe in Section 4.1. The rest of metadata is bitmap. Alternatively, the bitmap size can be set arbitrarily large to have node degrees greater than 55, as shown in Fig. 1b. If we use metadata larger than an 8-byte word, explicit bitmap logging becomes necessary because the bitmap that spans across multiple 8-byte words cannot be modified atomically. That is, `word B` in Fig. 1b can be unexpectedly flushed to PM while we are updating `word A` in CPU cache. To prevent such premature flush, we may employ hardware transactional memory to implement an atomic multi-word write function; however, hardware transactional memory incurs additional overhead [18] and requires specialized hardware. We note that the node size depends on the dimension of MBRs. With two-dimensional and three-dimensional MBRs, the node size with 8-byte metadata is about 2 KB and 4 KB, respectively.

3.2 Failure-Atomic Insertion

When inserting a new spatial object into an R-tree, the R-tree is traversed recursively from the root node to a leaf node. At each node, a candidate child node is selected using a legacy heuristic, such as the *least enlargement algorithm* [13]. If the chosen MBR does not completely overlap the new spatial object, the MBR must be enlarged to contain the new object, as shown in Fig. 2a. In legacy disk-based R-trees, an MBR update requires expensive per-node logging of the entire tree node to provide atomicity and crash consistency. That is, all updated nodes are duplicated as log entries, as shown in Fig. 2b and then checkpointed to the R-tree index file later.

However, in byte-addressable PM, the per-node logging, i.e., duplication of dirty data, incurs unnecessary memory copy overhead. Instead of relying on expensive logging, FBR-tree carefully enforces the order of store instructions to preserve the invariants of R-tree index such that it guarantees the failure-atomicity.

Algorithm 1 shows the insertion algorithm of FBR-tree. First, we select a child node and enlarge the MBR of the child node, as shown in Fig. 2a. Then, we call `mfence` and `clflush` to persist the updated MBR. Note that we perform in-place updates for the selected MBR (step `1` in the example) without logging. An MBR has multiple spatial coordinates, thus it cannot be updated atomically. To guarantee failure-atomicity, we may copy-on-write the MBR such that we retain the old MBR while writing a new one. Then, we validate the new copy by atomically flipping the valid bit of the old copy and the new copy. However, we note that such a CoW is not necessary for MBR updates because insertions only enlarge the size of the MBRs. In other words, as long as the MBR contains all MBRs of child nodes, MBR updates do not have to be atomic and the

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.

write ordering of spatial coordinates does not violate the correctness of the index. For example, no matter whether the boundary of the first dimension or the second dimension is updated, the partially updated MBR will still include all child MBRs. Suppose a system crashes after only one of the boundaries is overwritten. Subsequent queries will still successfully find and visit the child node if their query ranges overlap the child node's MBR.

Algorithm 1. Insert(Obj Obj, Node *N, Node *Parent)

```
n → mutex.lock()
if node is NOT leaf then
  locked = true;
  pos = PickChild(obj.r, n);
  n → child[pos].mbr = Combine(n → child[pos].mbr, obj.r);
  persist(n → child[pos]); // step 1 in Fig. 2a
  if n → child[pos] is NOT FULL then
    n → mutex.unlock(); locked = false;
  end if
  c_sibling = Insert(obj, n → child[pos].ptr, parent);
  if c_sibling is NOT NULL then
    child node has split, c_sibling is its sibling node
    Rect c_sibling_mbr = getMBR(sibling)
    if n is also full then
      create a splitLog (parent, current and a new sibling)
      persist(splitLog);
      sibling = split(n, c_sibling, parent);
      n → mutex.unlock();
    else
      n → child[free].mbr = c_sibling_mbr;
      persist(n → child[free]); // step 4 in Fig. 4.
      n → child[pos].mbr = getMBR(n → child[pos]);
      increase n → version and update valid bit of free;
      persist(n → metadata); // step 5 in Fig. 4.
      persist(n → child[pos].mbr); // step 6 in Fig. 4.
      persist(n → child[pos].ptr→version = 1); // step 7
      sibling = NULL;
      n → mutex.unlock();
    end if
  else
    sibling = NULL;
    if locked is true then
      n → mutex.unlock()
    end if
  end if
  return sibling;
else
  if n → bitmap is FULL then
    create a splitLog (parent, current and sibling)
    sibling = split(n, obj, parent);
  else
    pos = n → getFreeSpace()
    n → child[pos] = obj;
    persist(n → child[pos]); // step 2 in Fig. 2(a)
    increase n → version and update valid bit of pos;
    persist(n → metadata); // step 3 in Fig. 2(a)
    sibling = NULL;
  end if
  n → mutex.unlock();
  return sibling;
end if
```

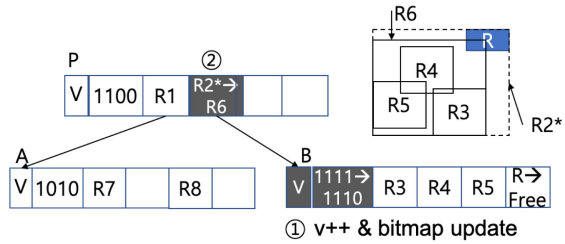


Fig. 3. Byte-addressable deletion in FBR-tree.

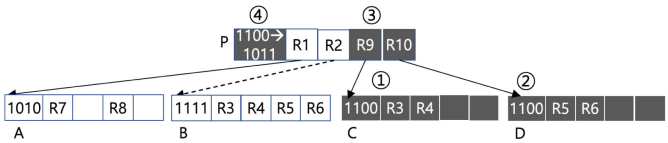
We note that partially enlarged MBRs can hurt the efficiency of the index because the partially enlarged MBR may unnecessarily overlap incoming queries due to the *dead space*, i.e., space that contains no data objects. However, such false positive results do not hurt the correctness, and the partially updated MBRs never return false negative results. We also note that such a dead space problem is not permanent since a dead space can be removed when a node splits or when underutilized nodes merge.

On the way down to a leaf node, we keep updating MBRs when necessary so that all ancestor nodes contain the new spatial object. Once we find a leaf node and insert a new spatial object, we search for free space by checking the bitmap in the leaf node and store the object's spatial coordinates in it (step 2 in Fig. 2a). Then, we call `mfence` and `clflush` to persist the new object. In the next step (step 3 in Fig. 2a), we increase the version number and update the bitmap to validate the new object. If the version number and bitmap are stored in an 8-byte word, they can be atomically updated and flushed. The version number update is necessary to enable a lock-free search, which we will discuss in Section 4.1. If a system crashes before the bitmap is updated, the written spatial object will be ignored and considered as a free space when the system recovers, i.e., no recovery process is required. In such a sense, the insertion algorithm of the FBR-tree is failure-atomic although it does not perform logging.

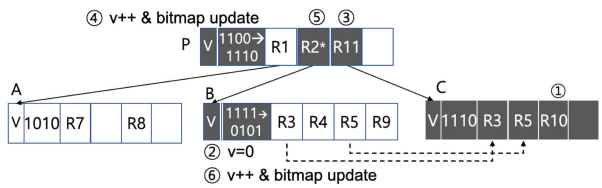
3.3 Failure-Atomic Deletion

When an indexed spatial object is deleted from a tree node, FBR-tree flips the valid bit of the object and flushes the bitmap to persist it, as shown in the first step of Fig. 3. If the deleted object is entirely within the MBR of its leaf node, the MBR of the leaf node will not be modified. However, if the deleted object shares at least one boundary with the MBR of leaf node, the MBR of the leaf node needs to be shruken by the deletion. In such a case, we backtrack to the parent node and update the leaf node's MBR accordingly. To reconstruct the MBR, we select the minimum and maximum boundaries in each dimension and perform in-place updates to overwrite the existing MBR. Again, we note that shrinking the MBR also does not have to be atomic because partially updated MBR does not affect the invariants of the index. In other words, no matter what dimension has been updated and flushed to PM, when a system crashes, the partially updated MBR will still contain all valid spatial objects in the sub-tree, and it will guarantee correct search results. Therefore, the FBR-tree deletion algorithm is also failure-atomic and guarantees consistency. Note that such in-place updates greatly reduce the amount of I/O since transactions do not need to perform expensive logging for each deletion or insertion.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.



(a) Byte-addressable CoW Split



(b) Byte-addressable In-Place Split

Fig. 4. Order of memory writes for node split in FBR-tree.

3.4 Failure-Atomic Page Split

Insertions and deletions often result in node overflows and underflows, which respectively require nodes to split and merge such that the tree height becomes re-balanced. In disk-based R-trees and B-tree variants [4], [56], *per-node* logging or journaling has been used because multiple tree nodes including a parent node need to be updated atomically. In legacy logging or journaling, unmodified portions of tree nodes are duplicated in a log or journal file because the minimum write granularity of disks is a disk page. Such legacy disk-based logging not only increases the write traffic but also blocks concurrent access to tree nodes. Unlike a disk-based R-tree, FBR-tree re-balances the tree height in a byte-addressable and failure-atomic manner without explicit logging.

3.4.1 Byte-Addressable Copy-on-Write Split

Fig. 4a shows the steps required to perform byte-addressable CoW when a node overflows. First, we allocate two new nodes (node `C` and `D`) and copy half of the entries from an overflow node to these nodes (steps 1 and 2 in the example). We note that this is not different from legacy CoW. However, unlike disk-based CoW, which can atomically update the parent node via disk-based block I/O, the entire parent node cannot be atomically updated in byte-addressable PM. Hence, we need to carefully enforce the ordering of 8-byte store and `clflush` instructions to update the parent node in a failure-atomic manner. Once we add the new MBRs and pointers to the parent node `P` (step 3 in the example), we overwrite the bitmap via a single store instruction and call `clflush` to persist it.

If a system crashes before we add the new nodes to the parent, the index is consistent because nothing has been changed to the parent node. If a system crashes after we add the two new nodes to the parent node but before we update the bitmap, the index is still consistent because the two child nodes will be considered invalid, i.e., free spaces. PM heap manager, such as Intel's PMDK [20] or HPE's NVMM [30] should deal with memory leak problems. I.e., a PM heap manager should create a log when it allocates a PM block. Then, the heap manager can check whether each PM block is being used by an application when a system recovers. If not, the block must be garbage collected. Note that this is not a requirement specific to our FBR-tree. All other PM-based data structures also require this feature to prevent memory leak problems. If a system crashes after the bitmap is updated, the index is in a consistent state; thus, recovery is not required.

CoW split updates three bits, two bits to validate new nodes and another bit to invalidate the overflow node. If the size of a bitmap is larger than 8 bytes, CoW split does not provide failure-atomicity because three bits can be in different words. Therefore, the maximum number of child nodes we can store in each tree node is limited to 55. If we want a tree node to have a larger number of child nodes, as shown in Fig. 1b, we must use explicit logging to update a bitmap. However, the size of a bitmap log will still be much smaller than legacy per-node logging. If a system crashes before we put a commit mark in the bitmap log, the logged bitmap will be ignored, and the index will be in its previous consistent state. If a system crashes after we put a commit mark in the bitmap log, the index will be in a new consistent state. Therefore, the byte-addressable CoW split algorithm is failure-atomic.

A drawback of a CoW split is that we need at least two free spaces on the parent node. If there is only one last free space in the parent node, the parent node cannot accommodate two new child nodes; thus the parent node also has to split. For example, after we update the bitmap of parent node `P` in the example shown in Fig. 4a, `R2` will become a free space. However, if another child node splits again, node `P` must split even though there is a single free space. Such a premature split degrades node utilization.

3.4.2 Byte-Addressable In-Place Split

The byte-addressable CoW split algorithm creates a copy of all the MBRs of an overflow node, which we find inefficient. We can reduce the number of memory writes by reusing the overflow node and creating a single new node instead of two. Algorithm 2 shows the in-place split algorithm, and Fig. 4b shows the in-place split algorithm steps with an example.

First, we allocate memory space for a new sibling node (`C` in the example) and copy half of the entries to this sibling node. For the other half, we reuse the overflow node (`B` in the example). In the second step, we set the version of the overflow node to 0, which indicates that the node is splitting and its MBR in the parent node may not reflect the actual MBR of the node. Note that we have not added the new sibling node (`C`) to the parent node, nor have the migrated entries been invalidated in the overflow node, i.e., we have not updated the bitmap of overflow node. Therefore, subsequent transactions do not miss any child nodes and the correctness of search operations is not compromised when a node is in this state. In the next step (step 3), we add the address and MBR (`R11`) of the new sibling node (`C`) to the parent node. Note that the new MBR (`R11`) is not valid until its corresponding bitmap is updated in the next step (step 4). Since we store the version and bitmap in the same 8-byte word, the version and bitmap are atomically updated. After validating the new child node, we update the MBR of the overflow node to make it just small enough to include the remaining MBRs (step 5). The MBR of the overflow node must not be updated prior to validating a new child

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.

node. Otherwise, a query that searches for a migrated entry may fail to find it. For example, suppose a query is searching for R5 in Fig. 4b, which is migrated to node C. If we reduce the area of R2, before adding C to the parent node, R5 will not be included in the updated R2* and a concurrent query looking for R5 will not visit both node B and C. As a result it will fail to find R5. To avoid this problem, the ordering of each update must be strictly enforced. Therefore, we call mfence and clflush in each step to guarantee failure-atomicity and consistency. Finally, we increase the overflow node's version to indicate the split process has completed (step 6). At the same time, we update the bitmap to invalidate the node entries migrated to the right sibling node. Again, we note that the version and bitmap are atomically updated.

Algorithm 2. split_inplace(Node *N, Entry *new_entry)

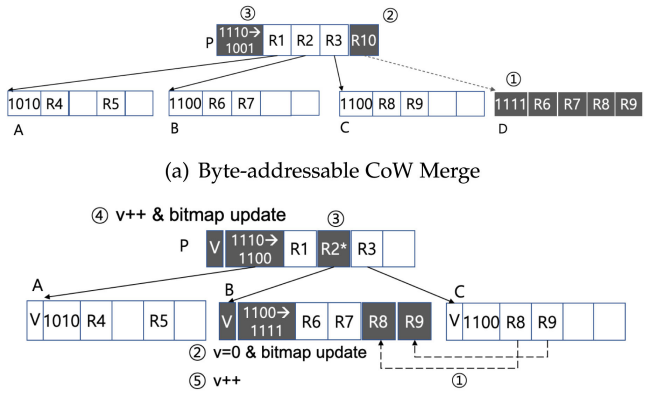
```
sibling = create a sibling node;
cluster node entries into group A and B
add new_entry to either A or B
n → version = 0; // indicate this node is splitting
persist(n → version);
tmp_metadata.bitmap = n→bitmap;
for i = 0; i < size; i++ do
  if n → branch[i] is in group B then
    addEntry(sibling, n → child[i]);
    tmp_metadata.bitmap[i] = 0;
  end if
end for
sibling → version = 1;
persist(sibling); // persist a new sibling node
tmp_metata.version = 1;
n → metadata = tmp_metdata;
persist(n → metadata); // persist bitmap and version
parent = pop();
if parent is full then
  if parent is root then
    root_split(parent, sibling);
  else
    split(parent, sibling);
  end if
else
  AddEntry(parent, sibling);
end if
```

3.5 Failure-Atomic Node Merge

If deleting a key from a node causes an underflow, FBR-tree redistributes entries between sibling nodes or merges two nodes if they fit in a single node. In this section, we compare two merge operation methods–(1) byte-addressable CoW merge and (2) byte-addressable in-place merge.

3.5.1 Byte-Addressable Copy-on-Write Merge

The byte-addressable CoW merge algorithm is similar to the byte-addressable CoW split, but it performs the memory writes in reverse order. Fig. 5a shows the steps required for a merge operation. In the walking example, we delete an entry from node B, which causes an underflow. Since the leaf node B cannot borrow an entry from its sibling node C, nodes B and C are to be merged. Therefore, we allocate a new node D and copy R6, R7, R8, and R9 to this new node. Then, we persist node D (step 1). In the next step, we add the MBR of node D to the parent node P as in normal insertion algorithm (step 2 and 3). Since the failure-atomic bitmap update will invalidate two under-utilized nodes and validate the new merged node atomically, the CoW merge algorithm is failure-atomic.



(a) Byte-addressable CoW Merge

(b) Byte-addressable In-Place Merge

Fig. 5. Order of memory writes for node merge in FBR-tree.

3.5.2 Byte-Addressable In-Place Merge

Differing from the CoW split, the CoW merge does not cause the node utilization problem. However, CoW operations require more memory copy operations than in-place updates. Thus, we design and implement the in-place merge algorithm.

As shown in Fig. 5b, when a node underflows, we copy all entries from a sibling node to the underflow node (step 1), and update the bitmap to validate the migrated entries (step 2). We set the version of node B to 0 to indicate that a rebalancing operation is updating the underflow node and its MBR in the parent node may not enclose all the MBRs of the node. In the next step (step 3), the MBR of node B in the parent node is updated to include all the entries migrated from an underutilized node. Then, the bitmap of the parent node and its version number are updated atomically to invalidate the underflow node C (step 4). Once the underflow node is removed from the parent node, the version of the merged node is increased (step 5) to indicate the merge operation is completed.

4 Concurrency and Consistency

4.1 Lock-Free Search

With the increasing prevalence of many-core systems, the importance of concurrent data structures also increases. One challenge for concurrent data structures is the lock contention between concurrent transactions. If a transaction accesses a data structure while it is being modified by another transaction, it may access the data structure in an inconsistent state and return incorrect results. Various lock methods have been used to protect data structures from concurrent accesses. However, due to synchronization overhead, lock methods often degrade the concurrency level and degrade performance. To reduce the synchronization overhead, various optimistic synchronization methods, i.e., lock-free search algorithms have been proposed in the

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.

literature [14], [18], [25]. With optimistic synchronization, search operations access data structures without acquiring shared locks. Instead, search operations optimistically access each node of the data structures and rollback when they later find that they have accessed inconsistent nodes.

Algorithm 3. Search(Node *N, Query *Q)

```
hitCount = 0;
if n != leaf then
  version = n → version;
  initialize child_queue;
  while true do
    for i=0; i < n → size; i++ do
      if branchOverlap(n→branch[i],q) && n→bitmap[i]==1
      then
        child_queue.push( n→branch[i] );
      end if
    end for
    if version==n→version or n→version==0 then
      – this node has not changed or it is being split
      while !child_queue.empty() do
        ret = Search(child_queue.front(), q);
        if ret = BACKTRACK then
          continue;
        end if
        child_queue.pop();
      end while
      break;
    else
      – this node has changed
      remove all entries in childqueue
      version = n→version;
    end if
  end while
else if n is leaf then
  version = n → version;
  initialize obj_queue;
  for i=0; i < size; i++ do
    if branchOverlap(n→branch[i],q) && n→bitmap[i]==1
    then
      – add an overlapping object to a result set
    end if
  end for
  if version==n→version then
    – this node has not changed
    return a result set
  else
    – this node has changed
    – remove objects found in this node from the result set
    – backtrack to parent node and restart search
    return BACKTRACK;
  end if
end if
return resultSet
```

FBR-tree takes such an optimistic approach, and the non-blocking search operations of the FBR-tree guarantee system-wide progress and consistency. Each 8-byte store instruction executed by the FBR-tree update algorithms preserves the invariants of R-trees. Therefore, even if a write transaction is making changes to an FBR-tree that are partially updated, concurrent read transactions will find the invariants of FBR-tree remain unchanged. I.e., concurrent read transactions can construct a consistent view of tree nodes without waiting for write transactions to release the exclusive lock.

The detailed algorithm of lock-free search in FBR-tree is shown in Algorithm 3. In FBR-tree nodes, a version number is stored along with a bitmap in each node to indicate the node is in a transient inconsistent tree state, i.e., when an entry in a tree node is added or deleted, its version number is increased. When a search query visits a tree node, it remembers its version number. Later, when the query is done accessing the node, it verifies that the version number has not changed. If the version has changed, the query knows that the node has been modified. Then, the query reads the node again to reflect the modifications. We note that such version-based lock-free control is commonly used in numerous lock-free concurrent structures [15].

A rollback operation is expensive, as it may repeat reading the same node many times. However, as we will show in Section 5, the rollback operation occurs very rarely. Therefore, lock-free search with rollback operations is often less expensive than the reader-writer lock mechanism because a reader-writer lock requires read transactions perform memory writes for each node visit [15], [18]. Besides, a write transaction updates only a few cachelines in a tree node. Therefore, reading the same node often benefits from CPU cache hits.

4.2 Serializability

Let us consider the serializability of concurrent read and write transactions. We assume each write transaction inserts a single spatial object. If a write transaction inserts multiple objects into different leaf nodes, multidimensional range queries that perform backtracking in a lock-free manner may encounter *phantom reads* and work in *read uncommitted* mode [46]. That is, range queries may find some, but not all of the objects inserted by a concurrent write transaction.

Note that FBR-tree insertion, deletion, split, and merge algorithms do not allow lock-free writes. I.e., we use legacy exclusive locking to avoid write-write conflicts. This is because the FBR-tree must guarantee not only byte-addressable consistency but also durability as an additional challenge.

4.2.1 Non-Splitting Insertion

Let us consider the most straightforward case first – a write transaction that does not split a node. Consider the example shown in Fig. 2a. Suppose a read transaction accesses node B while another write transaction attempts to store MBR R into the same node. If the read transaction reads the node prior to its bitmap being updated, the read transaction returns without reading R because the bitmap is not updated. In this case, the two transactions are serializable without incurring any consistency issue (read → write). If the write transaction updates the bitmap before the read transaction returns, our search algorithm makes the read transaction access the node again. Thus, it will read the new MBR R. In this case, the two transactions are also serializable (write → read). We omit a discussion of deletions due to its symmetry with insertions.

4.2.2 Byte-Addressable CoW Split

Now, consider the byte-addressable CoW split using the example shown in Fig. 4a. Suppose a read transaction

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.

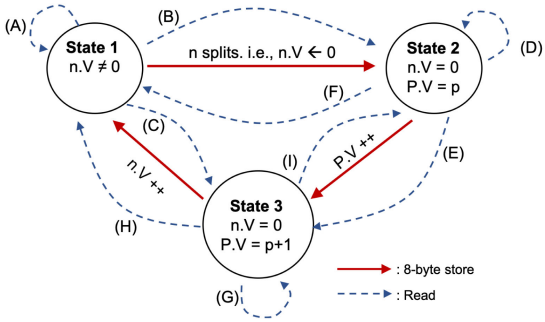


Fig. 6. State transitions with concurrent in-place updates (n: overfull node, P: parent node).

accesses node B and returns to its parent node P while another write transaction is making changes to P using CoW. If the version of node P is not yet changed, the read transaction does not have to read the new spatial object written by the write transaction. Thus, the read transaction will successfully return, and the two transactions are serializable (read → write). If the write transaction has already updated the bitmap of node P, the read transaction will discard the results found in leaf node B and visit new child nodes C and D. Again, the two transactions are serializable (write → read).

4.2.3 Byte-Addressable in-Place Split: Proof Sketch

As for the in-place split, we update the version of overflow node twice as shown in Fig. 4b. Therefore, the serializability for the in-place split is rather complicated.

Fig. 6 shows the state transition diagram with concurrent in-place updates and read queries, i.e., it shows all possible execution orderings of concurrent read and write transactions. The solid arrow indicates the version change of an overfull node and its parent node. In FBR-tree, whenever a tree node is updated, its version number is updated. The dashed arrow indicates the version change of a node—the node version when a read transaction accesses the first MBR in the node, and the version when it is done with accessing all the MBRs of the node.

i) Case A in Fig. 6 is trivial. If a read transaction accesses an overfull node and returns to its parent node prior to the overfull node's version being set to zero, the transactions are serializable (read → write). If the version of overfull node (n.V) is not 0, then the read query checks whether the version numbers have been changed. If they have been changed, the query reads the nodes again to guarantee serializability (write → read).

ii) In the case of B, a read transaction starts accessing an overfull node before a write transaction splits it. When the read transaction is done with the overfull node, it finds out that the overfull node is now being split. However, since the version of overfull node (n.V) is 0, not 1, the overfull node has not deleted any migrated entries. That is, the read transaction has processed all the sub-trees of the overfull node, and it will return correct search results. Therefore, the read transaction does not need to access its new sibling node and guarantees serializability (read → write).

iii) Case C also guarantees serializability (read → write). In case of C, a query visits an overfull node after it verifies the version of its parent node, as shown in Algorithm 3. Even if the overfull node splits and a new sibling node is added to the parent node, the query will not detect the split and return the same search results as in the case when the node has not split.

iv) In the case of D, a read transaction accesses an overfull node that is being split. Although a new sibling node could have been created, the overfull node has not deleted migrated entries. I.e., the in-place split algorithm deletes migrated entries when we set the version (n.V) to one. Therefore, the read transaction ignores its new sibling node and backtracks to the parent node, which guarantees the serializability (read → write).

v) Case E is similar to C. I.e., a read transaction verifies the version of the parent node before it visits its overfull child node. Therefore, the read transaction will not detect a new sibling node and return the same search results with the case when the node has not split. As such, it guarantees serializability (read → write).

vi) Case F is the opposite case of B. I.e., a read transaction starts accessing an overfull node while it is being split. When the read transaction is done with accessing the overfull node, it will find out the overfull node has completed the split. Therefore, the read transaction could have missed some sub-trees that were migrated to its new sibling node. To guarantee correct search results, our search algorithm makes the query backtrack to the parent node and restart tree traversals from the parent node.

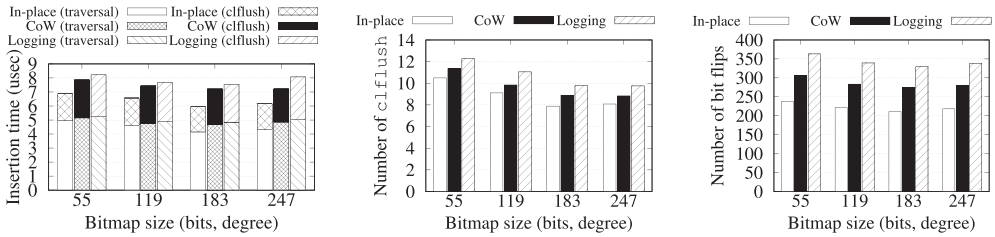
vii) In the case of G, a read transaction accesses an overfull node after its new sibling node has already been added to the parent node. I.e., if the query range overlaps the MBR of the new sibling node, the read transaction will access the right sibling node when it backtracks to the parent node. I.e., it guarantees serializability (write → read). Although the MBR of the overfull node could have been shrunken in the parent node, the read transaction may access the sub-trees that are migrated to the new sibling node due to the overlap between the overfull node and the right sibling node. Therefore, some sub-trees can be visited multiple times, which is not necessary. To avoid unnecessary multiple visits to the same sub-trees, we make read transactions to track the versions of visited tree nodes. Alternatively, read transactions may perform deduplication, which may hurt the search performance due to redundant node visits.

viii) In the case of H, a read transaction will detect the version of an overfull node is changed while it is reading the MBRs in the node. Since the read transaction could have missed some sub-trees that were migrated to the sibling node, the read transaction has to rollback and read the MBRs of the overfull node again to guarantee serializability (write → read).

ix) The case I does not occur in FBR-tree since the version of a tree node monotonically increases except when a node splits. Even if both child and parent nodes split recursively, the overfull child and parent nodes will keep the entries migrated to their new sibling nodes until they finish splits. Therefore, concurrent read transactions can access those nodes in a non-blocking manner.

We note that serializability is optional in many scientific applications and in database OLAP transactions [46]. If an application does not require serializability and strong consistency, our search algorithm may omit to check the

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.



(a) Insertion Time (b) Number of clflush (c) Number of bit flips

Fig. 7. Insertion performance with varying node size (AVG. of 5 Runs). We varied the degree of tree nodes by increasing the bitmap size. A larger node degree reduces the tree height at the cost of increased processing time of each node, which offsets the performance gain obtained by reducing the tree height.

version metadata and rollback operations to improve the concurrent query performance.

4.3 Recoverability

If a byte-addressable data structure enables lock-free search with persistent memory, the recovery process can be greatly simplified when a system crashes [18], [38]. I.e., even if a system crashes while a write transaction is making changes to an FBR-tree, subsequent transactions will find the partially written FBR-tree preserves the invariants of FBR-tree, and it can construct a consistent view of FBR-tree. The only corner case a recovery process needs to take care of is the node whose version is 0, which means rebalancing operation has not completed when a system crashes. Suppose a system crashes before the version of an overfull node is set to zero (state 1 in Fig. 6). Since nothing has been changed in the index, recovery is trivial. If a recovery process finds a node's version is 0, it has to determine whether it is in state 2 or 3. If a node is in state 2, its MBR in the parent node will match the actual MBR or the overfull node. Also, the overfull node will not have any duplicate sub-trees with its sibling node. If the recovery process finds a node is in state 2, it recovers from the failure by simply setting the version of the overfull node to a positive number. If a node is in state 3, its MBR in the parent node may not match the actual MBR of the overfull node, and there must be a sibling node with the same sub-trees with the overfull node. To recover from state 3, the recovery process invalidates duplicate sub-trees by updating the overfull node's version and bitmap.

5 Evaluation

We designed and implemented variants of FBR-trees and evaluate their performance on a workstation that has two Intel Xeon Gold 6230 processors (20 cores, 2.1 GHz, 20 x 32 KB instruction cache, 20 x 32 KB data cache, 20 x 1024 KB L2 cache, and 27.5 MB L3 cache), 375 GB of DDR4 DRAM and 732 GB Intel Optane DC Persistent Memory (DCPM). To create and manage FBR-trees in DCPM, we use Persistent Memory Development Kit (PMDK), which is designed by Intel to facilitate programming for persistent memory. To make use of atomic 8-byte instructions, we allocate a single large pool for each index and call `pmemobj_alloc()` for each tree node inside the pool, which returns an 8-byte offset to the node in the pool. For failure-atomicity, we carefully enforce the ordering of `mfence` and `clwb` instruction rather than using the PMDK transaction APIs, which performs expensive logging. We note that providing a directory name as the pool path allows the pool to dynamically create memory-mapped files, which removes the user-imposed limit on the size of the pool.

In the evaluation experiments, we used two datasets. One is a time series multidimensional *taxi service trajectory* dataset that has more than 80 million polylines and a total of nine attributes.[1] Since the real dataset is not large enough to sufficiently utilize 732 GB persistent memory of our testbed, we generate a synthetic 80 GB 3D point datasets in random distribution. For both datasets, we generated synthetic range queries simulating a varying number of users posing queries to the index, modeled as a Poisson process. The workload generator creates range queries from various synthetic distributions such as uniform, zipfian, and Z-order curve. We present the performance results of query workloads in uniform distribution and Z-order curve, but the results of other workloads are not significantly different.

When inserting spatial objects, our implementation of FBR-tree uses the least enlargement algorithm [13] to select a child node. Although there exist numerous hueristics when selecting a child node, we note that those optimizations are irrelevant to the byte-addressability and persistency of FBR-tree, and FBR-tree can employ such optimizations to improve the efficiency of index.

5.1 Byte-Addressable CoW Split Versus In-Place Split

In the first set of experiments shown in Figs. 7, 8, and 9, we insert 80 million Taxi trajectory polylines into an FBR-tree and measure the average query latency while increasing the size of bitmaps, i.e., the node degree. Unlike disk-based data structures, each node size does not have to match the disk block size. I.e., the tree node size is a performance tuning parameter that can be set arbitrarily in in-memory data structures. In disk-based R-trees, the degree of a node decreases as the dimension increases, which aggravates the well-known *curse of dimensionality* problem. However, the FBR-tree node size does not have to match the disk block size. Thus, the degree of a node in FBR-tree nodes is independent of the dimension. When the bitmap size is 7 bytes, i.e., when the degree of a node is 55, FBR-tree does not require bitmap logging when rebalancing because the bitmap can be atomically updated via 8-byte store instruction. For larger bitmap sizes, FBR-tree requires bitmap logging because the bitmap that spans multiple words cannot be updated atomically without logging.

1. The dataset is available at https://archive.ics.uci.edu/ml/

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.

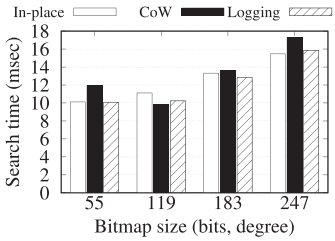


Fig. 8. Search performance with varying rebalancing methods.

In the experiments, we compare the performance of three rebalancing methods. `CoW` denotes the performance of byte-addressable copy-on-write split, `In-place` denotes the performance of byte-addressable in-place split, and `Logging` denotes the performance of legacy per-node logging. `(traversal)` denotes the tree traversal time, which includes the MBR computations and LLC misses caused by node visits, and `(clflush)` denotes the time to update PM, i.e., the overhead of `pmemobj_alloc()`, `store`, `mfence`, and `clflush` instructions.

Fig. 7 shows the performance of insert queries. Overall, the non-byte-addressable legacy `Logging`, demonstrates the worst performance because it duplicates entire dirty nodes to the logging space, which is dynamically allocated and deallocated. `CoW` also shows worse performance than `In-place` but better performance than `Logging`. This is because `CoW` does not duplicate the parent node when a node splits, but `Logging` creates a copy of the parent node. When a node splits, `In-place` calls `pmemobj_alloc()` only once for a new sibling node. However, `CoW` calls `pmemobj_alloc()` twice for two new nodes, and `Logging` calls the `pmemobj_alloc()` once more for the parent node update. Therefore, the number of `clflush` required for `Logging` is approximately 12 and 23 percent greater than that of `In-place` and `CoW`, respectively. In particular, `In-place` is up to 19.8 percent faster than `CoW` in terms of node update time `(clflush)`. Relative to tree traversal time `(traversal)`, `In-place` is also approximately 5.1 percent faster than `CoW` because `In-place` invalidates cachelines from CPU caches less frequently than `CoW` and `Logging`. Therefore, `In-place` spends less time selecting a child node.

The `CoW` split algorithm's side effect is that it leaves free space in the middle of an array because the old entry for an overflowing child node is marked as free space. Such fragmentation can degrade the tree node lookup performance. Because of the free space in the middle, the ordering of MBRs in each tree node can be different depending on whether we use `CoW` or `In-place`, which accounts for the difference in traversal time.

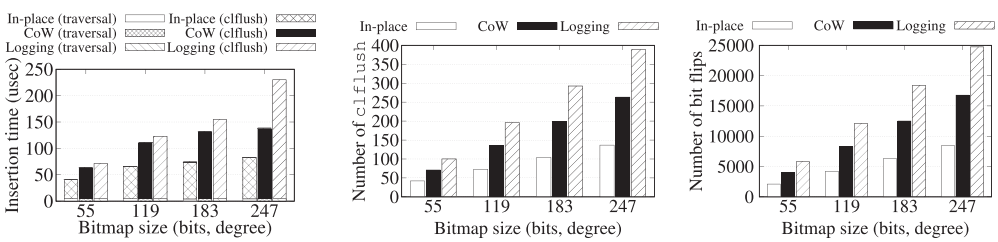
Fig. 7c shows the number of bit flips. PM technologies only support a limited number of writes per cell, and bit flipping consumes most of the power required for PM; thus, the number of bit flipping is an important performance metric in PM systems. The results demonstrate that `In-place` reduces the number of bits flipped by up to 23 and 36 percent over `CoW` and `Logging`, respectively.

Fig. 8 shows the range query performance. With larger bitmap sizes, the node degree increases and tree height is reduced. However, a large number of node degree requires more comparisons against MBRs in each node; therefore, search performance degrades as the degree increases. Although the search performance in read-only workloads is slightly affected by `CoW` and `In-Place` split algorithms, it is known that the search performance is highly dependent on various heuristic MBR reduction algorithms rather `CoW` or `In-place` [13]. We note that such heuristic MBR reduction techniques are irrelevant to the byte-addressable persistent memory, and beyond the scope of this work. For the rest of the experiments, we set the bitmap size to 23 bytes (183 bits) because this gave the fastest insertion performance with all three schemes.

Fig. 9 shows the performance of *splitting* insert queries, i.e., the queries that result in node splits. In particular, the rebalancing methods make a significant difference in node split performance. We note that Fig. 7 shows how different node split algorithms affect the average insertion performance, i.e., the split overhead is averaged over all other insert queries that do not split nodes. Since a node split modifies more than three nodes, it calls a much larger number of `clflush` instructions than the average. When the node degree is 247, the legacy logging scheme flushes more than 388 cachelines. Due to such a large number of cacheline flushes, the time taken to flush cachelines accounts for over 94 percent (94 percent for `In-Place`, 96 percent for `CoW`, and 97 percent for `Logging`). The node split time increases linearly as the degree of node increases. However, it is noteworthy that the overall average insertion performance, shown in Fig. 7a does not change significantly because the frequency of node split decreases as the node size increases.

5.2 Index Size Effect

In the experiments shown in Fig. 10, we generated a very large number of synthetic 3D points in uniform distribution and varied the number of indexed data points to evaluate the scalability of FBR-tree. Overall, `In-place` consistently



(a) Splitting Insertion Time (b) Number of clflush per Overflow (c) Number of bit flips

Fig. 9. Performance of insert queries that split overfull nodes. In-place updates reduces the number of cacheline flushes and improves the node split performance.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.

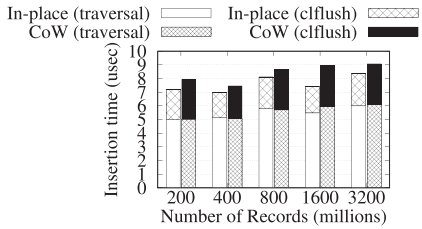


Fig. 10. Insertion performance with varying index size.

outperforms CoW by 6.5∼17 percent. If we insert 3200 million records, both CoW and In-place create FBR-trees with a total size of about 170 GB. Insertion of 3200 million records into an empty FBR-tree on our testbed machine takes about 7.4 and 8 hours for CoW and In-place, respectively. However, we observe that the average insertion time is rather insensitive to the index size because the insertion time depends on the tree height, which increases on a log scale, but not by the index size. Fig. 10 shows the average insertion time increases only by 45 and 51 percent for CoW and In-place as we increase the number of indexed data from 200 million to 3200 million.

5.3 Concurrency and Recoverability

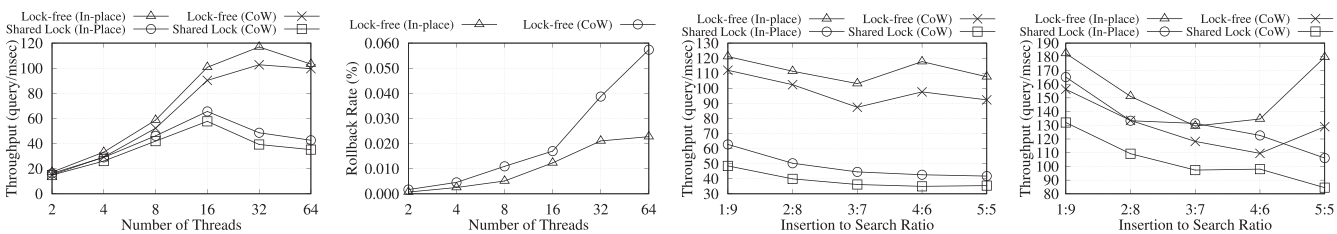
In the experiments shown in Fig. 11, we evaluate the performance of multi-threaded FBR-trees on Optane DCPM. We implemented the lock-free search algorithm we described in Section 4.1 and the *crabbing* protocol [46] as a baseline, which uses `std::shared_mutex` class in C++17.

In the experiments shown in Fig. 11a, we increase the number of concurrent threads that submit insert and search queries into FBR-trees with 1 million Taxi trajectory polylines. Each thread alternates between three insert queries and seven search queries. We use a relatively small dataset for this experiment because lock-contention occurs more frequently when the index is small. As the number of concurrent threads increases, the throughput of lock-free implementations improves up to 32 threads while `std::shared_mutex` implementation does not scale well over 16 threads due to the lock contention. Note that the crabbing protocol we use for insert transactions must hold an exclusive lock on a parent node until its child node splits, or it determines that its child node has sufficient free space such that a split is not required. Therefore, due to lock contention in upper-level tree structures, both CoW and In-place that use the crabbing protocol do not scale well and they fail to benefit from high parallelism. This result shows that lock contention becomes a dominant performance factor rather than the memory access operations. In contrast, the lock-free implementations of CoW and In-place gain up to 2.6x and 2.4x higher throughputs than the crabbing protocol versions, respectively. We note that lock-free read algorithm also suffers from concurrent access to the same node, i.e., it performs rollback operations if it detects a node version change. However, Fig. 11b shows the probability of rollbacks, i.e., the number of rollback operations divided by the number of visited nodes, is as low as 0.06 percent when 64 threads are concurrently accessing the same FBR-tree.

It is noteworthy that the performance of lock-free implementations becomes saturated when the number of threads exceeds the number of cores in a single socket, which is because of NUMA effects. It has been reported that NUMA effects for Optane DCPM are much more significant than they are for DRAM [1]. We note that our testbed machine has 6 interleaved DCPM's across two NUMA sockets. With interleaved DCPMs, we believe there is not much that user-level applications can do to avoid NUMA effects. However, with non-interleaved DCPMs, user-level applications may create a separate pool for each NUMA node and decide which pool to use. However, we note that NUMA locality in a hierarchical tree structure is a hard problem. Suppose a tree node splits and we need to decide which pool to use for the new node. If we place the new tree node in the same NUMA node with the overfull node, the whole tree structure will reside in a single NUMA node. If we place the new tree node in a different NUMA node in a round-robin fashion, it will not make much difference from the case when we use interleaved DCPMs. We leave the optimization of hierrachical tree structures for non-interleaved PM pools as our future work.

In the experiments shown in Fig. 11c, we run 40 threads that alternate submitting a different number of insert and search queries. We note that the range query execution time is much higher than the insertion time. Although a read transaction does not call clflush instructions, a multidimensional range query traverses a much larger number of tree nodes up and down because there can be multiple child nodes that overlap a given query. On the contrary, the number of nodes that a write transaction visits is limited by the tree height.

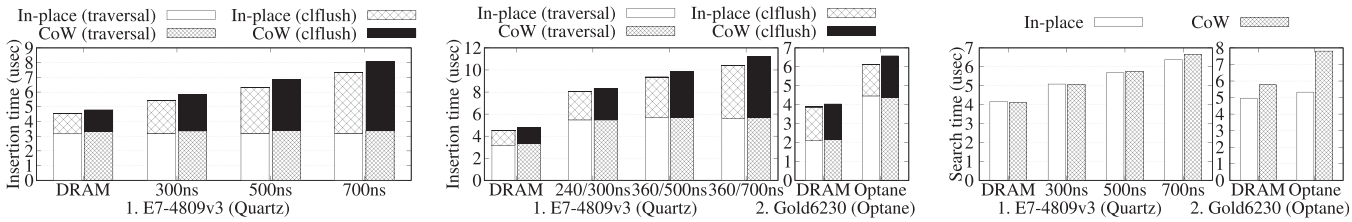
Similar to the results shown in Fig. 11a, the shared lock implementations suffer more from lock contention as we submit more insert queries. Although write transactions in lock-free search implementations also use exclusive write locks, search transactions can access tree nodes without acquiring shared locks. Therefore, the lock contention is much less serious than shared lock implementations. However, we note that a write transaction in lock-free search implementations



(a) Scalability of Multi-threaded FBR-tree (b) Probability of Rollback (c) Varying Insertion to Search Ratio (Uniform) (d) Varying Insertion to Search Ratio (Locality)

Fig. 11. Performance of multi-threaded FBR-tree.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.



(a) Average Insertion Time (Write Latency) (b) Average Insertion Time (R/W Latency) (c) Average Search Time (Read Latency)

Fig. 12. Performance with Varying PM Latency (AVG. of 5 Runs). We vary the PM latencies using the PM emulator on Intel Haswell processors and compare the performance against Optane DCPM on Intel Gold processors.

can be blocked if another thread's write transaction holds an exclusive lock. As our workload runs multiple threads that submit queries in a batch, the blocked write transaction also blocks subsequent read transactions in the same thread. As a result, as we submit more insert queries, lock contention gets worse and the throughput decreases.

The experiments shown in Fig. 11d are similar to those shown in Fig. 11c except that we submit clustered queries. That is, instead of the Taxi trajectory dataset, we generate synthetic 3D points using Z-order curve. Z-curve order is one of the well known space-filling curves that preserve spatial locality. If we insert data points in the order of Z-values, transactions are likely to access the same tree nodes and suffer more from lock contention. However, interestingly, Fig. 11d shows that the throughput of all implementations are higher than the Taxi trajectory dataset. This is because the Z-curve workload has a very high locality and all implementations benefit from CPU cache hits.

It is also noteworthy that the throughput of lock-free implementations improves if insert queries account for more than 40 percent. This is because range queries are much slower than insert queries in R-trees. I.e., each concurrent thread submits a fewer number of long running queries. Therefore, the throughput improves. However, if we submit a larger number of insert queries, the shared lock implementations suffer from a larger number of exclusive locks, and the throughput degrades.

Note that these experiments with multi-threaded FBR-trees not only evaluate the scalability but also show the instant recoverability of FBR-tree. We run a large number of search queries while write transactions, which make various tree nodes transiently inconsistent, are often suspended by the OS. A large number of read transactions concurrently access those partially updated nodes but return correct results. Even if a system crashes and partially updated tree nodes are persistently stored, concurrent read transactions can construct a consistent view of index because the invariants of FBR-tree are not violated. Therefore, read transactions can return correct search results. We also performed software error testing to validate the crash consistency of FBR-trees empirically and verified that partially updated FBR-tree nodes do not affect the invariants of index. It should be noted that the nodes with version 0 affect the performance, but they do not affect correctness since duplicate records can be detected and removed from result sets.

5.4 PM Latency Effect

Although Intel's Optane DCPM is on the market, it is not the only emerging byte-addressable persistent memory technology, but other emerging persistent memory technologies, such as STT-MRAM [16], PCM [54], and battery-backed NVDIMM are expected to offer a large performance spectrum [23]. Therefore, we use Quartz, a DRAM-based PM latency emulator [31], [51] to vary the PM latency when measuring the performance of FBR-tree. We note that Quartz has been used in numerous previous studies [3], [17], [18], [24], [33], [38], [41], [45], [52], [52]. Quartz models PM latency by inserting stall cycles at the boundaries of a small time interval called *epoch*. In our experiments, the minimum and maximum epochs are set to 5 and 10 nsec, respectively. We assume that PM bandwidth is the same as that of DRAM because Quartz does not allow us to emulate latency and bandwidth simultaneously.

We note that we run this experiments on a different testbed since Quartz is not supported by the latest 7th and 8th Intel Xeon processors, which is required by Optane DCPM, but only by old Haswell processors. Therefore, we run Quartz experiments on a workstation that has four Intel Xeon Haswell-EX E7-4809 v3 processors (8 cores, 2.0 GHz, 8 x 32 KB instruction cache, 8 x 32 KB data cache, 8 x 256 KB L2 cache, and 20 MB L3 cache) and 64 GB of DDR3 DRAM.

In the experiments shown in Fig. 12, we insert 80 million polylines in batches and breakdown the insertion time spent on each query as the read and write latencies of PM vary. In Fig. 12a, we set the read latency of PM to that of DRAM but increase the write latency. Therefore, tree traversal times are unaffected by PM write latency; however, the cacheline flush overhead increases as PM write latency is increased. `In-place` calls fewer cacheline flushes than `CoW`; thus, the performance gap between `In-place` and `CoW` is widened up to 9 percent due to the difference in flush overhead.

In the experiments shown in Fig. 12b, we vary both PM read and write latencies using Quartz on the Haswell processor testbed machine. We also run the same experiments using Optane DCPM on the other Gold processor testbed. The read latency of the local node memory in Haswell testbed machine is approximately 100 nsec. The average insertion time increases as we increase both read and write latencies. Interestingly, insertion performance is more sensitive to PM write latency than read latency due to the CPU cache effects.

In the experiments shown in Fig. 12c, we generate synthetic range queries in uniform distribution and submit 10,000 queries in a batch. The average selection ratio of the range queries is set to 1.9 percent. Note that we do not show the results of other selection ratios because no critical differences are observed. As we increase the read latency of PM, the query latency also increases; however, this does not result in a difference in the relative performance of the two split methods.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.

6 Conclusion

In this study, we have designed and implemented Failure-atomic Byte-addressable R-tree to obtain the most benefit from byte-addressability and the high-performance of PM. We carefully control the order of store and cacheline flush instructions and prevent single store instructions from making the FBR-tree inconsistent. Our performance study demonstrates that the FBR-tree reduces legacy logging overhead. In addition, the lock-free range query algorithm shows up to 2.6 times higher query processing throughput than the shared lock-based crabbing concurrency protocol. We also show that our FBR-tree on PM improves the performance of range query on HDF datasets by three orders of magnitude against the standard HDF-EOS range query functions.

Acknowledgments

This work was supported in part by the R&D program of National Research Foundation of Korea (NRF) under Grants NRF-2016M3C4A7952587 and NRF-2018R1A2B3006681, in part by IITP under Grant 2018-0-00549, and in part by Electronics and Telecommunications Research Institute (ETRI) under Grant 20ZS1310 funded by the Korean government.

References

[1] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th Conf. File Storage Technol.*, 2020, pp. 169–182.

[2] H. Andrade, T. Kurc, A. Sussman, and J. Saltz, "Efficient execution of multiple query workloads in data analysis applications," in *Proc. ACM/IEEE Conf. Supercomputing*, 2001, p. 34

[3] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 707–722.

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

[5] X. Chen, Y. Wang, E. Schoenfeld, M. M. Saltz, J. H. Saltz, and F. Wang, "Spatio-temporal analysis for New York state SPARCS data," in *Proc. Summit Clin. Res. Informat.*, 2017, pp. 483–492.

[6] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *Proc. 22nd ACM Symp. Operating Syst. Princ.*, 2009, pp. 133–146.

[7] P. Cudre-Mauroux, E. Wu, and S. Madden, "TrajStore: An adaptive storage system for very large trajectory data sets," in *Proc. IEEE 26th Int. Conf. Data Eng.*, 2010, pp. 109–120.

[8] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang, "High performance database logging using storage class memory," in *Proc. 27th Int. Conf. Data Eng.*, 2011, pp. 1221–1231.

[9] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi, "Trajectory pattern mining," in *Proc. 13th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2007, pp. 330–339.

[10] S. Goil and A. N. Choudhary, "High performance multidimensional analysis and data mining," in *Proc. ACM/IEEE Conf. Supercomputing*, 1998, pp. 21–21.

[11] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for GPU execution of tree traversals," in *Proc. Int. Conf. High Perf. Comput. Netw. Storage Anal.*, 2013, pp. 10:1–10:12.

[12] M. Gowanlock and H. Casanova, "Indexing of spatiotemporal trajectories for efficient distance threshold similarity searches on the GPU," in *Proc. 28th IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 387–396.

[13] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1984, pp. 47–57.

[14] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991.

[15] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann, 2008.

[16] Y. Huai, "Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects," *AAPPS Bull.*, vol. 18, no. 6, pp. 33–40, 2008.

[17] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-aware logging in transaction systems," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 389–400, 2014.

[18] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B+-trees," in *Proc. 16th USENIX Conf. File Storage*, 2018, pp. 187–200.

[19] Intel, "Intel and Micron produce breakthrough memory technology," 2018. [Online]. Available: https://newsroom.intel.com/news-releases/intel-and- micron-produce-breakthrough-memory-technology

[20] Intel, "PMDK: Persistent memory development kit," 2018. [Online]. Available: https://github.com/pmem/pmdk

[21] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via JUSTDO logging," in *Proc. 21st Int. Conf. Architectural Support Program. Lang.*, 2016, pp. 427–442.

[22] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen, "Discovery of convoys in trajectory databases," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 1068–1080, Aug. 2008.

[23] Y. Jin, M. Shihab, and M. Jung, "Area, power, and latency considerations of STT-MRAM to substitute for main memory," in *Proc. Memory Forum 41st Int. Symp. Comput. Architecture*, 2014.

[24] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "NVWAL: Exploiting NVRAM in write-ahead logging," in *Proc. 21st Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2016, pp. 385–398.

[25] A. Kogan and E. Petrank, "A method for creating fast wait-free data structures," in *Proc. 17th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2012, pp. 141–150.

[26] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proc. 21st Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2016, pp. 399–411.

[27] T. Kurc *et al.*, "A simulation and data analysis system for large scale, data-driven oil reservoir simulation studies," *Concurrency Comput.: Pract. Experience*, vol. 17, pp. 1441–1467, 2005.

[28] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz, "Querying very large multi-dimensional datasets in ADR," in *Proc. ACM/IEEE Conf. Supercomputing*, 1999, pp. 12–12.

[29] Y. Kwon, D. Nunley, J. P. Gardner, M. Balazinska, B. Howe, and S. Loebman, "Scalable clustering algorithm for N-body simulations in a shared-nothing cluster," in *Proc. 22nd Int. Conf. Sci. Statist. Database Manage.*, 2010, pp. 132–150.

[30] H. E. Lab, "Memory driven computing," 2018. [Online]. Available: https://www.labs.hpe.com/next-next/mdc

[31] H. E. Lab, "Quartz," 2018. [Online]. Available: https://github.com/HewlettPackard/quartz

[32] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 73–80.

[33] 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 *Proc. 15th USENIX Conf. File Storage Technol.*, 2017, pp. 257–270.

[34] Z. Li *et al.*, "MoveMine: Mining moving object data for discovery of animal movement patterns," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 4, pp. 37:1–37:32, Jul. 2011.

[35] D. Morozov and T. Peterka, "Efficient delaunay tessellation through K-D tree decomposition," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 728–738.

[36] B. Nam, H. Andrade, and A. Sussman, "Multiple range query optimization with distributed cache indexing," in *Proc. ACM/IEEE Conf. Supercomputing*, 2006, pp. 35–35.

[37] B. Nam and A. Sussman, "A comparative study of spatial indexing techniques for multidimensional scientific datasets," in *Proc. 16th Int. Conf. Sci. Statist. Database Manage.*, 2004, pp. 171–180.

[38] M. Nam, H. Cha, Y. Ri Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 31–44.

[39] G. Oh, S. Kim, S.-W. Lee, and B. Moon, "SQLite optimization with phase change memory for mobile applications," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1454–1465, 2015.

[40] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 12.

[41] 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 *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 371–386.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.

[42] V. Pascucci and R. J. Frank, "Global static indexing for real-time exploration of very large regular grids," in *Proc. ACM/IEEE Conf. Supercomputing*, 2001, pp. 45–45.

[43] A. Rudoff, "Programming models for emerging non-volatile memory technologies," *login*, vol. 38, no. 3, pp. 40–45, Jun. 2013.

[44] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on NVM," in *Proc. 31st Int. Conf. Massive Stroage Syst.*, 2015, pp. 1–14.

[45] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 91–104.

[46] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts*. New York, NY, USA: McGraw-Hill, 2005.

[47] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger, "Metadata efficiency in versioning file systems," in *Proc. 2nd USENIX Conf. File Storage Technol.*, 2003, pp. 43–58.

[48] F. Tauheed, L. Biveinis, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki, "Accelerating range queries for brain simulations," in *Proc. 28th Int. Conf. Data Eng.*, 2012, pp. 941–952.

[49] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, Art. no. 5.

[50] M. R. Vieira, P. Bakalov, and V. J. Tsotras, "On-line discovery of flock patterns in spatio-temporal data," in *Proc. 17th ACM SIGSPATIAL Int. Conf. Advances Geographic Inf. Syst.*, 2009, pp. 286–295.

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

[52] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proc. 16th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2011, pp. 91–104.

[53] M. S. Warren, "2HOT: An improved parallel hashed Oct-tree N-body algorithm for cosmological simulation," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, Art. no. 72.

[54] H.-S. P. Wong *et al.*, "Phase change memory," *Proc. IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.

[55] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 323–338.

[56] J. Yang, Q. Wei, C. Chen, C. Wang, and K. L. Yong, "NV-Tree: Reducing consistency const for NVM-based single level systems," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 167–181.

[57] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Proc. 31st Int. Conf. Massive Stroage Syst.*, 2015, pp. 1–10.

[58] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2013, pp. 421–432.

[59] P. Zuo and Y. Hua, "A write-friendly hashing scheme for non-volatile memory systems," in *Proc. 33st Int. Conf. Massive Storage Syst. Technol.*, 2017.

[60] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 461–476.

Soojeong Cho received the BS degree in computer science and engineering from Dankook University, Korea, in 2018. She is currently working toward the graduate degree with the School of Electrical and Computer Engineering, UNIST (Ulsan National Institute of Science and Technology), Korea. Her research interests include persistent memory and big data processing systems.

Wonbae Kim received the BS degree in computer science and engineering from the Ulsan National Institute of Science and Technology, Korea, in 2015. He is currently working toward the PhD degree with the School of Electrical and Computer Engineering, UNIST (Ulsan National Institute of Science and Technology), Korea. His research interests include big data processing systems, machine learning platforms, and persistent memory.

Sehyeon Oh received the BS degree in computer science and engineering from the Ulsan National Institute of Science and Technology, Korea, in 2018. He is currently working toward the graduate degree with the School of Electrical and Computer Engineering, UNIST (Ulsan National Institute of Science and Technology), Korea. His research interests include embedded database systems and persistent memory.

Changdae Kim received the BS, MS, and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), South Korea. He is a research fellow at ETRI. His research interests include computer architecture, operating systems, and cloud computing.

Kwangwon Koh received the MS degree in computer science from Yonsei University, South Korea, and the PhD degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2008. He is a research fellow at ETRI. He has worked on various projects including virtual machine monitor for ARM architecture, supercomputing system for genome processing, and scalable operating system for many-core processors. He currently focuses on the software stack of a multi-tier memory system. His research interests include system software for parallel computing, virtualization, and multi-tier memory system.

Beomseok Nam (Member, IEEE) received the BS and MS degrees from Seoul National University, South Korea, and the PhD degree in computer science from the University of Maryland, College Park, Maryland, in 2007. He is an associate professor at SungKyunKwan University, Korea. Before that he was an assistant/associate professor at UNIST (Ulsan National Institute of Science and Technology), South Korea. His research interests include data-intensive computing, database systems, and embedded system software. He is a member of IEEE, ACM, and USENIX.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on November 26,2021 at 06:17:25 UTC from IEEE Xplore. Restrictions apply.