clfB-tree: Cacheline Friendly Persistent B-tree for NVRAM

WOOK-HEE KIM, JIHYE SEO, JINWOONG KIM, and BEOMSEOK NAM,
Ulsan National Institute of Science and Technology

Emerging byte-addressable non-volatile memory (NVRAM) is expected to replace block device storages as an alternative low-latency persistent storage device. If NVRAM is used as a persistent storage device, a cache line instead of a disk page will be the unit of data transfer, consistency, and durability.

In this work, we design and develop *clfB-tree*—a B-tree structure whose tree node fits in a single cache line. We employ existing *write combining store buffer* and *restricted transactional memory* to provide a *failure-atomic cache line write* operation. Using the failure-atomic cache line write operations, we atomically update a clfB-tree node via a single cache line flush instruction without major changes in hardware. However, there exist many processors that do not provide SW interface for transactional memory. For those processors, our proposed clfB-tree achieves atomicity and consistency via in-place update, which requires maximum four cache line flushes. We evaluate the performance of clfB-tree on an NVRAM emulation board with ARM Cortex A-9 processor and a workstation that has Intel Xeon E7-4809 v3 processor. Our experimental results show clfB-tree outperforms wB-tree and CDDS B-tree by a large margin in terms of both insertion and search performance.

CCS Concepts: • **Information systems → Data structures**; **Storage class memory**;

Additional Key Words and Phrases: Non-volatile memory, data structure, persistent indexing

ACM Reference format:
Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. 2018. clfB-tree: Cacheline Friendly Persistent B-tree for NVRAM. *ACM Trans. Storage* 14, 1, Article 5 (February 2018), 17 pages.
https://doi.org/10.1145/3129263

1 INTRODUCTION

Recent advances of byte-addressable non-volatile memories (NVRAM), such as phase change memory and spin-transfer torque RAM, open up a new opportunity to reduce the I/O overhead of block granularity storage systems [11, 14]. However, the advent of the non-volatile memory places new challenges on the existing softwares, including B-trees, which is designed for volatile memories and block device storages that promise tri-atomic (never updated, complete, or unreadable) write operations for each disk page. For such block device storages, traditional shadowing or redo logging has been used to atomically persist changes to B-trees.

As NVRAM is expected to be accessible across a memory bus not via a PCI interface and the unit of atomicity in NVRAM is not 4KBytes page but a cache line, the biggest challenge for B-tree on NVRAM is how to atomically update B-tree nodes. In particular, there is no guarantee on when

Authors' addresses: W.-H. Kim, J. Seo, J. Kim, and B. Nam (corresponding author), Ulsan National Institute of Science and Technology (UNIST), School of Electrical and Computer Engineering, Ulsan, Ulsan, 44919, Republic of Korea; emails: {okie90, jihye, jwkim, bsnam}@unist.ac.kr.

https://doi.org/10.1145/3129263

the updates to B-tree nodes are written back to NVRAM as they can be cached in L1 or L2 caches and the memory controller can reorder writes at a cache line granularity. Although the ordering of memory writes can be preserved via cache line flush instructions, calling a large number of cache line flush instructions to enforce the ordering of memory writes is not a good idea, since the cache line flush is known to be very expensive [2, 7, 21, 25].

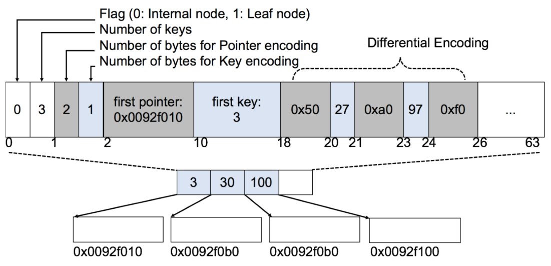
In this work, we propose *clfB-tree*—a cache line friendly B-tree whose internal tree node fits in a cache line. The size of a cache line depends on the memory access size, i.e., size of L1 and L2 caches (typically 32 or 64bytes). The tree node size of legacy B-tree is determined by disk page size, which can often hold more than hundreds of keys and child pointers. Such a high branching factor ensures few disk reads. However, if the tree node size is set to a cache line size, the degree of a tree node is bounded by a very small number, i.e., if the cache line size is 64bytes and keys and pointers are 8bytes, the degree of a tree node is just 4. Such a low degree of a tree node can make a tree height very high. To decrease the tree height, we propose *differential encoding* that keeps minimum number of bytes just enough to distinguish keys in the subtrees and thereby increasing the degree of a tree node.

By the nature of B-tree, a tree node often stores similar keys and pointers in it. As the level of tree node increases (closer to leaf nodes), the locality of keys increases; i.e., keys in a leaf node are similar. Also, heap managers often allocate contiguous memory blocks, which results in similar memory addresses. Leveraging these facts, clfB-tree computes and stores the differences between keys and pointers with fewer bytes instead of storing entire keys and pointers, which we will refer to as *differential encoding*. By taking advantage of the locality of keys and pointers in a tree node, the differential encoding can effectively compress the keys and the pointers and can increase the utilization of tree nodes.

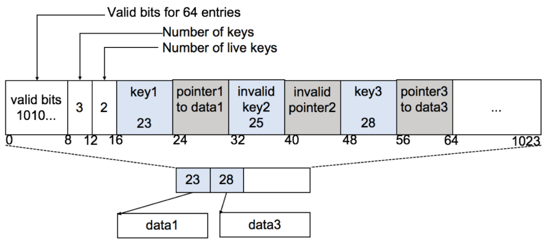
B-tree was originally designed and optimized for block device storage, thus it might not be the best indexing structure for NVRAM. If the size of B-tree node is larger than the cache line size, cache line flushes and memory fences must be carefully used to enforce the consistency of B-tree on NVRAM. Main memory data structures such as AVL-tree, Red-Black tree, or T-tree, whose branching factors are smaller than B-tree, can be considered an alternative indexing structure for NVRAM but those binary or tertiary trees are not good for cache locality, because each node occupies only a fraction of a cache line. Moreover, they require more frequent memory allocation than B-tree. `nvmalloc()` is expected to be much more expensive than DRAM `malloc()` because of the overhead of non-volatile namespace management [8]. Unlike these in-memory data structures, *clfB-tree* utilizes a cache line as an individual tree node. Therefore, it does not only require cache line padding but also simplifies persist operations.

The contributions of this work are as follows.

- **clfB-tree with Differential Encoding.** We design and develop *clfB-tree*—a B-tree structure whose tree node fits in a single cache line. Since a cache line is either 32bytes or 64bytes, it cannot hold a large number of entries in it. To store more entries in the small clfB-tree node, we propose differential encoding; we compute the differences between the first key/pointer and the other keys/pointers, and we store only the differences in the tree node. With the differential encoding, we can reduce the number of bytes used for each key and pointer and can store a larger number of entries in a tree node.
- **Failure-Atomic Cache Line Write.** We implement *failure-atomic cache line write* function explicitly using *restricted transactional memory* (RTM) and implicitly taking advantage of *write combining store buffer*. The failure-atomic cache line write function helps minimizing the number of cache line flushes so that only a single cache line flush instruction is required to update a tree node.


Flag (0: Internal node, 1: Leaf node)
Number of keys
Number of bytes for Pointer encoding
Number of bytes for Key encoding
Differential Encoding
first pointer: 0x0092f010
first key: 3
0x50
27
0xa0
97
0xf0
0x0092f010
0x0092f0b0
0x0092f0b0
0x0092f100

(a) Layout of clfB-Tree Internal Node



(b) Layout of clfB-Tree Leaf Node

Fig. 1. Node utilization improvement with differential encoding.

- **Analysis of Performance Implications of Various Atomic Writes.** For the processors that do not provide SW interface for transactional memory, we discuss performance implications of other atomic write methods for clfB-tree, including short-circuit shadowing, in-place update using per-cacheline log, and append-only update operation.

The rest of the article is organized as follows: In Section 2, we present our design and implementation of clfB-tree. In Section 3, we present how clfB-tree atomically updates its tree nodes using an atomic cache line write function that we implement using the restricted transactional memory and the write combining store buffer. In Section 4, we discuss how clfB-tree can update its tree nodes via in-place updates if processors do not support RTM. Section 5 presents the state-of-the-art wB-tree and compares it against clfB-tree. In Section 6, we present persistent heap management for clfB-tree. In Section 7, we present the performance results and analysis. Section 8 discusses other research efforts related to this study. Finally, we conclude in Section 9.

2 CLFB-TREE

2.1 Differential Encoding in Internal Node

Figure 1(a) shows the layout of a clfB-tree internal node. The size of a clfB-tree internal node is set to the cache line size (64bytes in the example). The first two bytes store metadata about the tree node. The first byte indicates whether the tree node is a leaf or an internal node and how many keys are in the tree node. The second byte stores the numbers of bytes used for encoded pointers and keys.

Suppose a clfB-tree node stores three keys—3, 30, and 100—and four child pointers, as illustrated in Figure 1(a). The 8byte pointer to the leftmost child node is stored after the 2bytes of metadata, and the smallest key 3 is stored next to it. For the rest of the pointers and the keys, we compute the differences against the first value and determine how many bytes are required to store the maximum difference. As for the keys, the values are monotonically increasing; thus, we can simply calculate and encode the difference between the first key and the last key. For the pointers, the memory addresses are not necessarily in an increasing order. Thus, we use the most significant bit as a sign bit for the encoding.

If pointer encoding and key encoding both require 1byte per each, then a 64byte tree node can have maximum 23 keys and 24 child pointers. We can further compress the keys, pointers, and metadata in the unit of a bit, but we found bitwise encoding requires more computations and it slows down overall indexing performance. In the worst case, although it is very unlikely, differential encoding still may need 8bytes for pointers and keys. If so, then a 64byte tree node can hold 3 keys and 4 child pointers at maximum.

If a key is string type and is larger than a cache line size, then we can store the common prefix of the string keys in a separate memory block and can store only the first different byte of the string keys in a clfB-tree node as in prefix B-tree [1].

2.2 Insertion to Leaf Node

We designed and implemented two variations of leaf nodes in clfB-tree. In the first implementation, which we refer to as *encoded leaf*, we set the size of a leaf node to the cache line size as in the internal node described above. A drawback of this approach is that a leaf node can not hold a large number of entries.

In the second implementation, we employ the append-only modification strategy as in wB+-tree [2] and NV-Tree [25]. In the *append-only leaf* version of clfB-tree, we append a new entry to the first available empty slot. Once the new entry is safely written back to NVRAM via a cache line flush and a store fence instruction, we atomically set the bit of the entry in a validness bitmap array to 1 and call another cache line flush and a store fence instruction. For delete and update operations, NV-tree appends a *negative* flag that invalidates an existing entry, which requires two cache line flush instructions as in insert operation. But wB+-tree and our clfB-tree simply invalidates an existing entry by flipping its validness bit in the bitmap array of the leaf node. This append-only approach was shown to be effective in reducing the number of cache line flushes [2, 25]. But an apparent drawback of this approach is that the keys are not sorted and binary search can not be used unless the tree node has additional information about the ordering. Thus, wB+-tree manages two metadata array, one for the validness *bitmap array* and the other for the ordering of entries—*slot-array*. A drawback of having two metadata arrays in a tree node is that it requires additional cache line flushes [2].

3 FAILURE-ATOMIC CACHE LINE UPDATES

3.1 Atomic Write Granularity

While atomic write operations have been supported in the granularity of 8bytes in modern processor design, a couple of previous works [4, 17] expect failure-atomic write instructions at a higher granularity will become available for NVRAM.

Modern processors have the *write combining store buffer* that buffers and combines multiple consecutive small 8byte writes. The processors atomically flush the dirty bytes to the memory system as a single transaction if they are multiple writes to the same cache line. However, the write combining store buffer currently does not support transactions. That is, even if there exist two consecutive `STORE` instructions for a cache line, other threads may cut in and evict the partially written dirty cache line. Thus, the write combining store buffer alone does not guarantee failure-atomic cache line writes. We expect a failure-atomic write instruction for a cache line can be easily implemented with small changes in current processor designs.

However, even without adding a new wider atomic write instruction to processors, we claim the failure-atomic cache line write function can be implemented by using *Restricted Transactional*

ALGORITHM 1: Failcure Atomic Write for Cache Line

```
procedure
atomicCacheLineWrite(src, dest)
while 1 do
    int status = _xbegin();
    if status == _XBEGIN_STARTED then
        memcpy(dest, src, 64);
        _xend();
        break;
    else
        ;
    end if
end while
clflush(dest)
```

Memory (RTM), which is available in Intel Haswell-EX architectures. RTM provides three new instructions—`XBEGIN`, `XEND`, and `XABORT`—that allow programmers to specify the start and the end of a transaction and to explicitly abort a transaction if the transaction cannot be successfully executed [5, 23]. RTM is originally designed to provide hardware transactional memory support for multi-threaded applications. However, if the RTM and the write combining store buffer are used together, we can achieve the desired behavior of the failure-atomic cache line write operation. Algorithm 1 shows our implementation of a failure-atomic cache line write function.

RTM guarantees the store operations in a transaction are not visible outside the transaction until `XEND` is successfully completed. That is, a dirty cache line (*dest*) remains temporarily in the write combining store buffer in the core that executes the *atomicCacheLineWrite*() function. If the system crashes before the transaction finishes, then the dirty cache line in L1 will be lost, and it does not hurt the consistency of clfB-tree.

The size of *dest* in Algorithm 1, even if we use RTM, should be no larger than a cache line size. If we modify more than one cache line inside an RTM transaction, then recently accessed cache lines can be in L2 or L3 caches but other cache lines can be flushed to NVRAM. This will violate failure-atomicity of write operation.

clfB-tree can take benefits of this wider atomic writes, since it can atomically update an entire tree node. Algorithm 2 shows how simple it is for a clfB-tree to atomically store two child pointers in a parent node using the `atomicCacheLineWrite`. To enforce the persistency of the change to a clfB-tree node, we still need to call a cache line flush and a store fence instruction, but only once, as shown in Figure 2(a).

4 CLFB-TREE UPDATES VIA 8 BYTES ATOMIC WRITES

Although the failure-atomic cache line write operation can be implemented via RTM and write combining store buffer, not all the processors support the hardware transactional memory extension. In this section, we discuss how clfB-tree can atomically update a tree node using *shadowing* or *in-place updates* instead of RTM.

4.1 Short-Circuit Shadowing

Shadowing is a copy-on-write technique that is widely being used to provide atomicity and durability in database systems. As shown in Figure 2(b), short-circuit shadowing [4] creates a copy of a tree node and modifies the copy. If the shadow copy is ready to be persisted, then the parent node

ALGORITHM 2: Atomic Insertion To Parent Node

```
procedure
$atomicInsert2ParentNode(btree, leftChild, rightChild, key)$
 1: $p \leftarrow btree.getParentNode()$;
 2: if $p.numEntries < MAX\_ENTRY$ then
 3:     $vp \leftarrow getVolatileInnerNode()$;
 4:     $vp \leftarrow p$;
 5:     $keys \leftarrow vp.insertKeyAndGetKeys(key)$;
 6:     $vp.numEntries \leftarrow vp.numEntries + 1$;
 7:     $ptrs \leftarrow vp.insertPtrAndGetPtrs(leftChild, rightChild)$;
 8:     $numBytes4Key \leftarrow getNumBytes4KeyEncoding(keys)$;
 9:     $numBytes4Ptr \leftarrow getNumBytes4PtrEncoding(ptrs)$;
10:     for $i \leftarrow 0, vp.numEntries - 1$ do
11:         $encPtr \leftarrow getDiff(ptrs[i+1], ptrs[0])$;
12:         $vp.storeEncPtr(i, encPtr)$;
13:         $encKey \leftarrow getDiff(keys[i+1], keys[0])$;
14:         $vp.storeEncKey(i, encKey)$;
15:     end for
16:     $atmoicCacheLineWrite(vp, p)$;
17:     $clflush4Node(p, p.numEntries)$;
18:     $mfence()$;
19: else
20:     $splitInnerNode(btree, leftChild, rightChild, key)$;
21: end if
end procedure
```

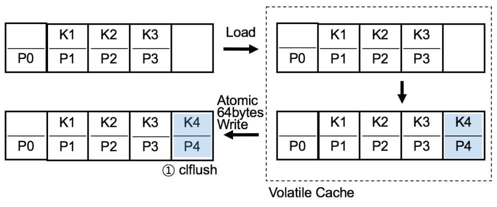
of the original node updates its child pointer via 8bytes of atomic write operation; the address of the shadow copy atomically replaces that of the original node. However, this short-circuit shadowing becomes complicated if a tree node splits and the parent node needs to update two pointers at the same time. To atomically update two pointers, we need to create another shadow copy of the parent node.

The short-circuit shadowing in clfB-tree requires two cache line flush instructions—one for the shadow node and the other for the parent node. clfB-tree needs only one cache line flush instruction for a shadow node, because the size of a tree node is the cache line size.

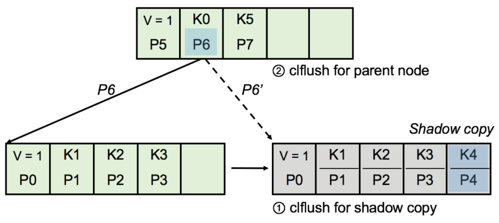
Unfortunately, the short-circuit shadowing in clfB-tree may result in cascading split problems, because a pointer to the shadow node may need more number of bytes than a pointer to the original node. If the memory address of the shadow copy is far from the memory address of the first child node, then the differential encoding may require more bytes. If so, then atomic 8byte write instruction cannot be used to replace the previous pointer. In such a case, instead, all the pointers in the parent node must be encoded again and the entire cache line needs to be updated. We refer to the problem as *upward cascading shadowing problem*. The upward cascading shadowing problem often occurs even if a leaf node does not overflow. Sometimes it may result in node split, which aggravates the problem even further.

4.2 In-Place Update

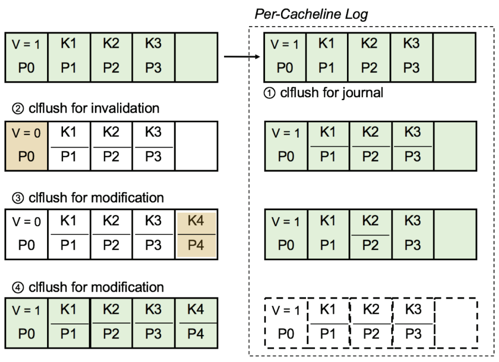
To resolve the upward cascading shadowing problem, *in-place update* can be employed to atomically update a clfB-tree node, as shown in Algorithm 3. The in-place update makes changes to an original tree node after creating a back-up copy. Since the encoded child node pointers in a



(a) Atomic Write



(b) Short-Circuit Shadowing


Per-Cacheline Log
① clflush for journal
② clflush for invalidation
③ clflush for modification
④ clflush for modification
V = 1
V = 0

(c) In-Place Update (atomic write granularity: 8 bytes)

Fig. 2. Atomic insert in clfB-tree.

parent node are not updated unless a child node splits, the in-place update does not suffer from the unnecessary upward cascading shadowing problem unless a leaf node overflows.

However, the *in-place update* requires four cache line flushes as shown in Figure 2(c): (i) First, when a new entry is stored in a tree node, a copy of the original tree node is saved as a *per-cacheline log*. This copy requires a cache line flush. (ii) In the next step, we invalidate the original node by setting its valid bit to *false*, which requires another cache line flush. (iii) After setting the valid bit to *false*, the original tree node can be updated by multiple write operations. If the system crashes in this step, then we can check the consistency of the tree node and recover from the failure by checking the valid bit and overwrite the original tree node with the saved per-cacheline log. After writing all the changes to the original tree node, we call another cache line flush to persist the changes. (iv) The last step is to validate the updated original tree node. Another cache line flush is necessary to flip the valit bit.

5 CLFB-TREE VS. WB+-TREE

In this section, we briefly discuss wB+-tree, which is the state of the art implementation of B+-tree for NVRAM and compare it with clfB-tree.

The *write-atomic B+-tree* (wB+-tree) [2] is an improved version of NV-tree [25]. Both wB+-tree and NV-tree reduce the number of cache line flushes by employing append-only updates to B-tree. A wB+-tree node needs two metadata—*bitmap array* and *slot array*. The bitmap array stores the locations of valid entries and the slot array stores the sorted order of the entries. The node structure of wB+-tree is illustrated in Figure 3(a).

When a new entry is inserted into a wB+-tree node, (i) it first invalidates the slot array by flipping a bit in the bitmap. This step requires a `clflush` instruction. (ii) Next, it stores the new

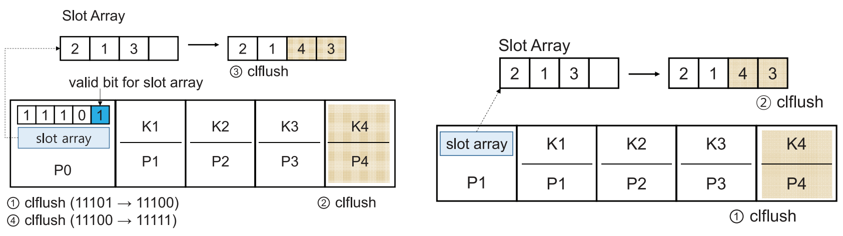
ALGORITHM 3: In-Place Parent Node Update

```
procedure
$inPlaceInsert2ParentNode(btree, leftChild, rightChild, key)$
 1: $p \leftarrow btree.getParentNode()$;
 2: $numBytes4Key \leftarrow getNumBytes4KeyEncoding(keys)$;
 3: $numBytes4Ptr \leftarrow getNumBytes4PtrEncoding(ptrs)$;
 4: if $p.numEntries < MAX\_ENTRY$ then
 5:     $perCacheLineLog \leftarrow nvMallocNode()$;
 6:     $memcpy(perCacheLineLog, p, CACHE\_LINE\_SIZE)$;
 7:     $mfence()$;
 8:     $clflush4Node(perCacheLineLog)$;
 9:     $p.valid = 0$;
10:     $mfence()$;
11:     $clflush(p.valid)$;
12:     $keys \leftarrow p.insertKeyAndGetKyes(key)$;
13:     $p.numEntries \leftarrow p.numEntries + 1$;
14:     $ptrs \leftarrow p.insertPtrAndGetPtrs(leftChild, rightChild)$;
15:     $mfence()$;
16:     $clflush4Node(p)$;
17:     $p.storeMetadata(ptr[0], key[0], numBytes4Key, numBytes4Ptr)$;
18:     for $i \leftarrow 0, numEntries - 1$ do
19:         $encPtr \leftarrow getDiff(ptrs[i+1] - ptrs[0])$;
20:         $p.storeEncPtr(i, encPtr)$;
21:         $encKey \leftarrow getDiff(keys[i+1] - keys[0])$;
22:         $p.storeEncKey(i, encKey)$;
23:     end for
24:     $mfence()$;
25:     $clflush4Node(p)$;
26:     $p.valid = 1$;
27:     $mfence()$;
28:     $clflush(p)$;
29:     $delete(perCacheLineLog)$;
30: else
31:     $splitInnerNode(btree, leftChild, rightChild, key)$;
32: end if
end procedure
```

entry in the first available slot, which requires another `clflush` instruction. (iii) Then, it updates the slot array with the updated order of entries. This step requires multiple `clflush` instructions as the size of slot array can be larger than the size of a cache line. (iv) Once the slot array is written back to NVRAM, it updates the bitmap array with one more `clflush` instruction so that the new entry and the slot array become valid.

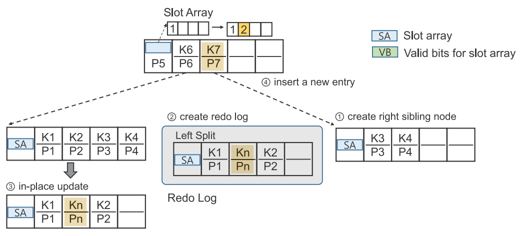
Update and delete operation are similar to insert operation; the changes to a tree node can be persisted by updating the slot array and bitmap array. With the slot array and the bitmap array, wB+-tree requires at least four cache line flushes per transaction.

A variant of wB+-tree is the *slot-only wB+-tree* that outperforms the *bitmap+slot wB+-tree* in terms of update response time, because it reduces the number of cache line flushes. The slot-only wB+-tree limits the maximum number of entries in a wB+-tree node; i.e., each tree node stores no more than seven entries. By limiting the number of entries, wB+-tree can combine the bitmap

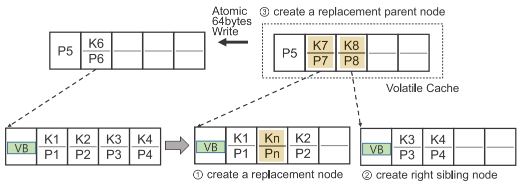


(a) wB-tree with Slot Array and Bitmap (b) wB-tree with Slot Array

Fig. 3. Atomic insert in wB+-tree.



(a) Leaf Split in wB+-tree


Atomic 64bytes Write
③ create a replacement parent node
Volatile Cache
① create a replacement node
② create right sibling node

(b) Leaf Split in clfB-tree

Fig. 4. Leaf split algorithm comparison.

array and the slot array, because the slot array can fit in 8bytes and the slot array can be atomically updated. By eliminating the bitmap array, the slot-only wB+-tree reduces the number of cache line flushes per insertion to two as long as a node does not split, as shown in Figure 3(b). When a new entry is stored in the slot-only wB+-tree node, (i) the entry is stored in the first available slot and `clflush` instruction is called. After the new entry is written to NVRAM, (ii) the slot array needs to be updated via another `clflush` instruction.

When a leaf node overflows, wB+-tree relies on *redo-logging* that calls at least two cache line flush instructions, as illustrated in Figure 4(a). When a leaf node splits, wB+-tree copies half of the entries from the overflown leaf to a new right sibling node and it inserts a pointer to the new sibling node in its parent node. Note that the update to the parent node and the update to the slot array of the overflown node must be performed atomically at the same time. If the system crashes before we update the slot array of the original node, then the original tree node and the new right sibling node may have redundant entries. To resolve this problem, wB+-tree employs redo-logging, which requires additional cache line flush operations.

Unlike wB+-tree, clfB-tree creates two new leaf nodes and copy half of the entries to each node when a leaf node overflows, as shown in Figure 4(b). After we persist the two new entries, we create a copy of the parent node in a volatile cache and store the two pointers in the parent node. Since the parent node can overwrite the existing parent node atomically via `atomicCacheLineWrite()` or *in-place parent update*, clfB-tree requires only three cache line flush operations for leaf node split as long as the size of a leaf node is a cache line size. If a leaf node is larger than the cache line size and an entire leaf node update requires k cache line flushes, then clfB-tree calls $2k + 1$ cache line flushes (1 for the parent node update, 2k for the two new nodes). In wB+-tree, a leaf node split requires $2k + 4$ cache line flushes (2 for the in-place update of the overflown node, k for the redo-logging, k for the right sibling node, and another 2 for the parent node update).

6 PERSISTENT MEMORY MANAGEMENT

Recently, various persistent heap managers and persistent file systems have been proposed for non-volatile memory management [3, 17, 21, 22, 24]. They provide various memory address binding schemes; NV-Heaps and Mnemosyne employ memory-mapped file [3, 22], but Heapo and SoftPM employ native heap [6, 8]. clfB-tree can employ either approach. But if clfB-tree is constructed as a file on a persistent file system, child pointer is no more a persistent memory address but a file offset, which can help significantly reduce the number of bytes used for child pointer encoding.

In this work, we implemented clfB-tree on top of Heapo [8]. Heapo is a heap-based persistent object store that defines persistent heap layout, namespace organization, object sharing, and protection mechanism. Heapo does not enforce memory persistence, but applications are instead responsible for persist barrier, memory barrier and cache line flush. Heapo implements static address binding as in Mnemosyne. Therefore, the memory address of a persistent object is not relocated.

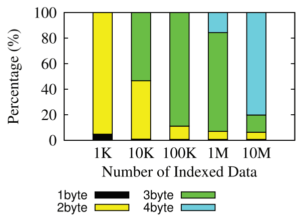
For each persistent clfB-tree, we first create a name space in Heapo. In the name space, Heapo applies the buddy algorithm to manage NVRAM blocks as in GLibc. B-tree insertion algorithm splits tree nodes in an irregular manner unless data are inserted in a sorted order. Thus, the memory addresses in a tree node are often discontiguous. However, although the memory addresses are not contiguous, the chances of a tree node's having similar pointers are very high, as we will show in Section 7.

7 EXPERIMENTS

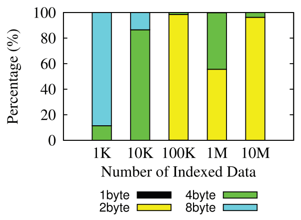
The goal of clfB-tree is to minimize the overhead of failure-atomic update on a B-tree. We evaluate clfB-tree to answer the following questions.

- Does the differential encoding improve node utilization?
- Does low cost of updating a small tree node recoup the performance loss incurred by the height of a tall tree? If not, then can we strike a balance between them via employing append-only leaf?
- How much overhead does cache line flush incur?
- Last, but most importantly, how fast is it?

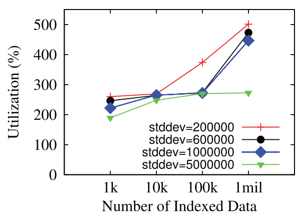
We implemented clfB-tree, slot-only wB-tree, and CDDS B-tree in C. We do not evaluate NV-tree [25] and FP-tree [19], since they do not guarantee the failure-atomicity of internal tree nodes. In other words, they are not persistent index, since they need to reconstruct entire tree structures from leaf nodes when systems fail. We integrated the three persistent B-trees with an NVRAM heap manager—*Heapo* [8]. We do not evaluate the bitmap+slot wB-tree, since slot-only wB-tree is known to be faster than bitmap+slot wB-tree. We run the experiments on two platforms. One is a workstation that has an Intel Xeon E7-4809 v3 processor (2.00GHz, $8 \times$ 32KB instruction caches, $8 \times$ 32KB data caches, $8 \times$ 256KBytes L2 caches, and 20MBytes L3 cache) and 64GB of DDR3 DRAM. Xeon E7-4809 v3 processor supports TSX (Transactional Synchronization Extensions) including RTM feature. In this platform, we assume NVRAM is as fast as DRAM and a specific address range



(a) Memory Allocation Locality with Varying the Number of Indexed Data



(b) Indexed Key Locality with Varying the Number of Indexed Data



(c) Node Utilization with Varying the Number of Indexed Data

Fig. 5. Node utilization improvement with differential encoding.

of DRAM is NVRAM. The other platform is an NVRAM emulation board, *Tuna* [13], that has ARM Cortex-A9 processor, Xilinx Zynq XC7Z020 ARM-FPGA SoC, and FPGA programmable logic that controls read/write latency of one of two DRAMs to emulate NVRAM latency.

7.1 Memory Allocation Locality

In the first experiments, we insert 10 million random key value pairs of long type into a clfB-tree. Figure 5(a) shows how many bytes are required for encoding each child pointer (persistent memory address obtained from Heapo) as we increase the number of inserted data. In the experiments, we set the size of an append-only leaf node to 512bytes, and the size of an internal node to 64bytes.

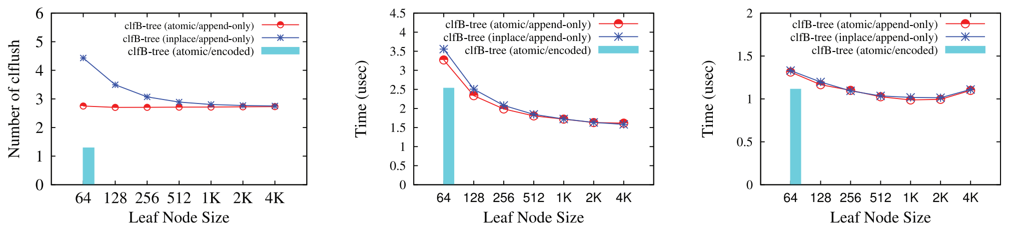
As we insert more data, tree size becomes larger, and we need more memory blocks. Therefore, the persistent memory addresses become distant from each other, and the differential encoding needs more bytes for the pointers. However, the differential encoding provides high compression rate, as shown in Figure 5(a). When we insert 10 million key-value pairs, not a single encoding uses more than 4bytes; 80.2% of the encodings use 4bytes and 19.8% use less than 4bytes. When we insert 100 thousands key-value pairs, the pointer encodings use at most 3bytes; 88.9% of the encodings use 3bytes and 11.1% use less than 3bytes.

Figure 5(b) shows the opposite results. When the number of inserted data is small, the keys in a clfB-tree node are often sparse and the key encodings use more bytes. As we insert more data, the keys, especially in leaf nodes, become denser and use less bytes for the encoding. When we index 1000 key value pairs, most of the pointer encodings use 2bytes, but most of the key encodings use 4bytes. On the contrary, when we index 10 million key value pairs, most of the pointer encodings use 4bytes, but most of the key encodings use 2bytes.

It should be noted that the key compression rate is dependent on key distribution. In the experiments shown Figure 5(c), we vary the number of inserted data and the standard deviation of the key distribution. As the standard deviation is high and the keys are sparse, the compression rate decreases and the node utilization drops. However, overall, we observe the node utilization of clfB-tree (number of encoded entries/number of available slots without encoding) is between 200% and 500%. It should be noted that B-tree node is not often fully utilized. That is, it is well known that the average node utilization of classic B-tree is 66%. Compared to that number, the differential encoding helps improve the node utilization by 3~7 times. To further improve the memory allocation locality, we can pre-allocate a large number of consecutive pages in advance. However, such a preallocation requires information about how much data will be stored in the future.

7.2 Update Overhead vs. Tree Height

In the experiments shown in Figure 6, we insert 1 million key value pairs of long type with varying the leaf node size. In *atomic/encoded* scheme, the leaf node size and the internal node size are fixed



(a) Average Number of clflush (b) Average Insertion Time (c) Average Search Time

Fig. 6. Performance with varying size of leaf nodes (AVG of 5 runs, 1 million data, Xeon E7-4809 v3).

Table 1. Number of Cache Line Flushes in Atomic/Append-Only (1 Million Insertions)

Leaf Size (bytes)	Inner Node		Leaf Node		Sum
	Split	No Split	Split	No Split	
64	545578	205332	1091156	908844	2750910
128	207060	82618	828240	1585880	2703798
256	95390	39796	763120	1809220	2707526
512	46204	19258	739264	1907592	2712318

Table 2. Number of Cache Line Flushes in In-Place/Append-Only (1 Million Insertions)

Leaf Size (bytes)	Inner Node		Leaf Node		Sum
	Split	No Split	Split	No Split	
64	2182312	239990	1091156	908844	4422302
128	828240	92224	828240	1585880	3334584
256	381560	45804	763120	1809220	2999704
512	184816	22034	739264	1907592	2853706

to 64bytes cache line size. For each insertion, we call the proposed failure-atomic cache line write function; a single `clflush` instruction is called for a leaf node update, and one-thirds of insertions cause a leaf node to split, which requires another `clflush` instruction for its parent node update.

In the append-only leaf node, denoted as *atomic/append-only* and *in-place/append-only*, it can hold no more than 511 entries, since the bitmap array cannot be larger than 64bytes. Figure 6(a) shows the append-only leaf node requires at least twice larger number of `clflush` instructions than *atomic/encoded*. When leaf node size is smaller, node split occurs more frequently. If parent nodes are updated via in-place update scheme, denoted as *in-place/append-only*, then each node split issues at least four cache line flushes and the number of `clflush` instructions sharply increases. However, if we update parent nodes using failure-atomic cache line write function, denoted as *atomic/append-only*, the parent node is updated by a single `clflush` instruction. The leaf node size in *atomic/append-only* scheme does not significantly affect the number of `clflush` instructions in *atomic/append-only* scheme because one additional `clflush` instruction is masked by a fewer number of `clflush` instructions required for splitting a small node. Tables 2 and 1 show how many cache lines are flushed for each different type of tree node udpates.

When the leaf size is 64bytes, the atomic encoded leaf scheme (*atomic/encoded*) calls much less cache line flushes than the append-only leaf scheme. Also, the *atomic/encoded* scheme stores more entries in its leaf nodes and makes the tree height smaller, thus its average insertion time

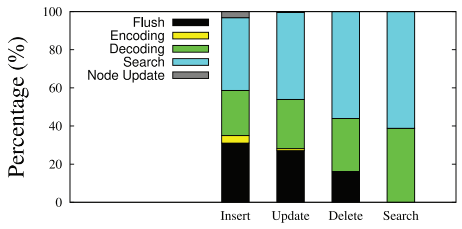


Fig. 7. Breakdown of time spent on each query type (AVG of 5 runs, atomic parent update/append-only, leaf size:1KB, Xeon E7-4809 v3).

is 29% and 41% lower than that of the append-only leaf schemes (2.53 vs. 3.27usec, 3.56usec), as shown in Figure 6(b). This result validates our proposition that the differential encoding helps improve the indexing performance. However, as we increase the leaf size of append-only leaf schemes, the tree heights of the append-only leaf schemes decrease and their insertion times outperform the *atomic/encoded* scheme although they need more cache line flushes. The different numbers of `clflush` in *in-place/append-only* and *atomic/append-only* result in a 10% difference in insertion time. As we will explain below, the overall overhead of cache line flush accounts for about 30% of insertion time.

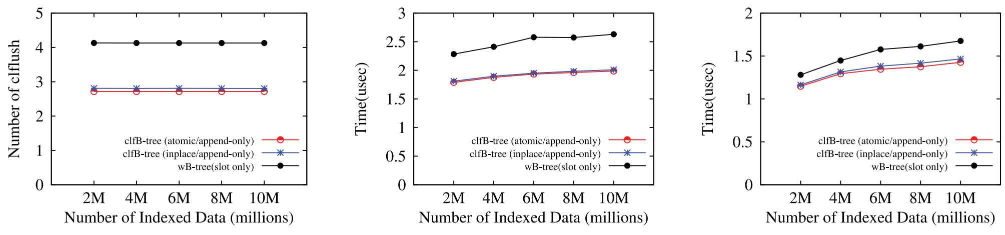
In the experiments shown in Figure 6(c), we submit one million queries and search a clfB-tree index that has 1 million key-value pairs. Note that the append-only leaf in clfB-tree does not sort the entries. Therefore, we need to linearly scan all the entries to find a matching key. Although a large leaf node decreases the tree height, it slows down the linear search performance when a leaf node is larger than 1KByte. Also, if a leaf node is too small, i.e., smaller than 1KByte, its search performance degrades, because the tree height becomes taller.

When the leaf size is set to the cache line size, the encoded leaf has more entries than the append-only leaf schemes. Therefore, the encoded leaf processes queries faster than append-only leaf schemes because of short search path length. However, when the size of append-only leaf node is between 256bytes and 2KBytes, the append-only leaf schemes perform search operation faster than the atomic encoded leaf scheme because of also short search path. Interestingly, the in-place parent update performs search query slightly slower than the atomic parent update. This is because the in-place parent update uses one byte for the valid bit, which decreases the node utilization and hurts the search performance.

For the rest of the experiments, we show the performance of the append-only 1KByte leaf schemes, because it shows the faster search and insertion performance than the *atomic/encoded* scheme.

7.3 Cache Line Flush Overhead

Figure 7 shows the breakdown of the time spent on each type of a query. Cache line flush accounts for about 31% of the insertion time in the *atomic/append-only* scheme of clfB-tree, while it takes about 33% of the insertion time in the *in-place/append-only* scheme. Since we reduce the cache line flush overhead enough so that other overhead in clfB-tree, such as differential encoding and decoding, can be more expensive than the overhead of the NVRAM write back operation. The overhead of cache line flushes for delete operation and update operation is smaller than the overhead of insert operation, since delete operation just flips a single bit in the bitmap array and the update operation does not split a tree node.



(a) Number of clflush (b) Average Insertion Time (c) Average Search Time

Fig. 8. Performance with varying number of indexed data (AVG of 5 runs, leaf size:1KB, Xeon E7-4809 v3).

7.4 Comparison Against wB+-Tree

Finally, we compare the performance of clfB-tree against the slot only wB-tree with varying the number of indexed data. Figure 8(a) shows the number of indexed data does not significantly affect the average number of cache line flushes per insertion, because most insertions modify only a single leaf node in the append-only leaf schemes. As discussed in Section 5, an insertion into a leaf node in the slot-only wB-tree calls two cache line flush instructions. However, if a leaf node splits, it calls a large number of cache line flush instructions for redo-logging, creating a sibling leaf node, and performing the in-place update. The leaf split algorithm in clfB-tree also calls a set of cache line flush instructions for creating two leaf nodes but it does not perform redo-logging. As a result, clfB-trees with append-only leaf nodes call about 2.7 cache line flush instructions per insertion on average, while wB-tree calls about 4.1 cache line instructions per insertion.

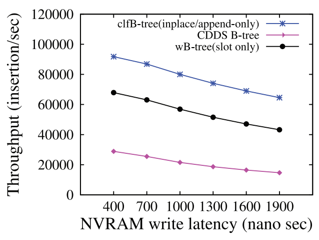
Figure 8(b) shows that the average insertion time increases as we insert more data, not because of more cache line flushes but because of the taller tree height. The *atomic/append-only* clfB-tree consistently outperforms the slot-only wB-tree by 28~33% and the performance gap increases as we index more data and the tree height grows.

As for the search performance, clfB-tree is also 11~17% faster than the wB-tree as shown in Figure 8(c). The search performance difference comes from the slow linear search on unsorted entries in large internal nodes of wB-tree.

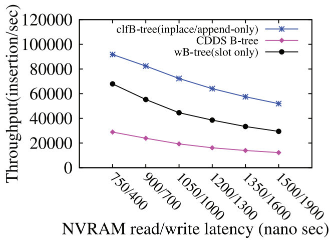
7.5 NVRAM Latency Effect

Now, we measure the sensitivity of persistent B-trees to NVRAM latency. Tuna [13], an NVRAM emulation board, allows us to adjust the read and write latency of DRAM; the write latency can be adjusted between 400ns and 2000ns, and the read latency can be adjusted between 750ns and 1500ns. Since ARM Cortex-A9 processor on Tuna board does not support RTM, we do not show the performance of *atomic/append-only* clfB-tree, but we only compare the performance of *in-place/append-only* clfB-tree against wB-tree and CDDS B-tree. The cache line size of ARM Cortex-A9 is 32bytes and the size of word size of Cortex-A9 is 4bytes. So, we assume that the unit of failure-atomicity writes is 4bytes, and 32byte cache lines can be flushed in a failure-atomic way. Therefore, we set the size of clfB-tree nodes to 32bytes. Hence, the memory address locality and the number of entries in each clfB-tree node are not very different from the experiments presented in the previous subsection. For the experiments on Tuna, we set the leaf node size to 512bytes.

Figure 9 shows the throughput of persistent trees when varying the latency of NVRAM on Tuna board. The write latency of NVRAM is expected to be much higher than DRAM latency. For example, phase change memory is about 2~5 times slower than DRAM as it has access latencies in the hundreds of nanoseconds [3]. Since the range of NVRAM read and write latency Tuna board supports is limited between 400nsec and 1,900nsec, we set the read latency to 750nsec and vary



(a) Varying Write Latency



(b) Varying Read/Write Latency

Fig. 9. Insertion throughput with varying latency (AVG of 5 runs, leaf size: 512bytes, 1 million, Tuna).

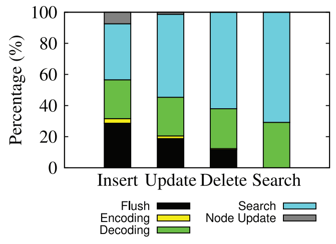


Fig. 10. Breakdown of time spent on each query type (AVG of 5 runs, in-place parent update/append-only, leaf size: 512bytes, Tuna).

the write latency in the experiments shown in Figure 9(a). As we increase the write latency, the insertion throughput of CDDS, clfB-tree (*in-place/append-only*), and the slot-only wB-tree decrease slightly.

In the experiments shown in Figure 9(b), we vary both read and write latencies of NVRAM and measure the insertion throughput. To navigate tree structures, all three persistent B-trees access a couple of internal tree nodes and all three indexing structures suffer from high read latency of the emulated NVRAM. Hence, the insertion throughput decreases linearly as we increase the latencies. Overall, clfB-tree consistently outperforms the slot-only wB-tree and the CDDS B-tree by a large margin in various NVRAM latency configurations.

8 RELATED WORK

The constraints on the ordering of NVRAM writes must be observed for the correctness of applications. To resolve the ordering issues of memory writes in NVRAM-based systems, numerous recent works [4, 16, 20] proposed novel memory persistency models, such as *strict persistency* [20] that does not distinguish memory consistency with memory persistency, and *epoch persistency* [4] that requires persist barriers so that persist order may deviate from the volatile memory order. Our work is different from theirs in that we devise failure-atomic cache line write function using RTM and write combining store buffer without changing processor designs.

There exist many recent works [2–4, 12, 16, 17, 21, 22, 25] that contributed to NVRAM management, which we believe complements our work. BPFS [4] is a transactional file system for NVRAM that adopts a short-circuit shadow paging using *epoch* barrier that specifies an ordering of groups of persist operations. However, the *epoch* barrier instruction requires a modification of memory hierarchy. NV-Heap [3] provides a way of managing NVRAM without difficult reasoning about thread safety, atomicity, and memory access ordering. It also uses epoch barriers and *mmap* to allow programmers to allocate, read and write, and deallocate objects in NVRAM. Mnemosyne [22] is a program interface that provides a set of persist primitives that consists of `mfence` and `clflush` for managing data objects in NVRAM. *Heapo* [8] is a memory allocator for NVRAM that provides programmers with simple but robust memory management interfaces. *NVMalloc* [17] is another NVRAM memory allocator that prevents memory wear-out and helps avoid erroneous memory writes and permanent system corruption. NVMDuet [15] proposed a memory scheduling policy for NVRAM, which resolves the contention between only persistent and non-persistent writes. These works are complementary to our work as clfB-tree can be deployed on top of strict persistency memory architectures, NVRAM heap managers, and so on.

As for the data consistency issues of persistent data structures, there exist several recent studies. SQLite/PPL [18] is a persistent per-page logging scheme for SQLite, which stores the per-page logs in PCM memory. NV-Logging proposed by Huang et al. [7] proposes per-transaction logging method for OLTP systems using NVRAM to avoid the bottleneck of centralized log buffers. They compared their per-transaction logging against *NV-Disk* [9, 10]—NVRAM used as as disk replacement accessed via standard file I/O interface—and showed that replacing disk or flash memory with NVRAM without redesigning file system cache and IO interface suffers from the high overhead [7].

The most similar related works to ours are CDDS, NV-Tree, and wB-tree [2, 21, 25]. *CDDS* [21] is a version-based data structure for NVRAM. CDDS allows atomic updates on NVRAM without logging. *NV-tree* [25] is a B-tree designed for NVRAM that reduces the number of cache line flush operations via append-only updates in leaf nodes. *wB-tree* [2] is an improved version of NV-tree as it employs append-only updates not only for leaf nodes but also for internal nodes.

9 CONCLUSION

In this work, we design and develop a cache line friendly persistent *clfB-tree*. To atomically persist each clfB-tree node via a single cache line flush, we develop fail-safe atomic cache line write function using the restricted transactional memory and the write combining store buffer. We also present in-place atomic update scheme for clfB-tree for the processors that do not support the restricted transactional memory. Our experimental results show that clfB-tree outperforms slot-only wB-tree in terms of insertion and search performance.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their suggestions and comments on the early draft of this article. This research was supported by MKE/KEIT (Grant No.10041608, Embedded System Software for New Memory based Smart Devices), MSIP (Grant No. R0190-15-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development), National Research Foundation of Korea (No. 2014R1A1A2058843 and No. NRF-2016M3C4A7952634, Development of Machine Learning Framework for Peta Flops Scale).

REFERENCES

[1] Rudolf Bayer and Karl Unterauer. 1977. Prefix B-trees. *ACM Trans. Database Syst. (TODS)* 2, 1 (1977), 11–26.
[2] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in non-volatile main memory. *Proc. VLDB Endow. (PVLDB)* 8, 7 (2015), 786–797.
[3] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persisten objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11).*
[4] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09).*
[5] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys'14).* 15:1–15:15.
[6] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. 2012. Software persistent memory. In *Proceedings of the USENIX Annual Technical Conference (ATC'12).*
[7] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware logging in transaction systems. *Proc. VLDB Endow.* 8, 4 (2014).
[8] Taeho Hwang, Jaemin Jung, and Youjip Won. 2014. HEAPO: Heap-based persistent object store. *ACM Trans. Stor. (TOS)* 11, 1 (2014).

[9] Dohee Kim, Eunji Lee, Sungyong Ahn, and Hyokyung Bahn. 2013. Improving the storage performance of smartphones through journaling in non-volatile memory. *IEEE Trans. Consum. Electron.* 59, 3 (2013), 556–561.

[10] Junghoon Kim, Changwoo Min, and Young Ik Eom. 2014. Reducing excessive journaling overhead with small-sized NVRAM for mobile devices. *IEEE Trans. Consum. Electron.* 6, 2 (June 2014).

[11] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youjip Won. 2014. Resolving journling of journal anomaly in android I/O: Multi-version B-tree with lazy split. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'14).*

[12] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. 2013. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST'13).*

[13] Taemin Lee, Dongki Kim, Hyunsun Park, and Sungjoo Yoo. 2014. FPGA-based prototyping systems for emerging memory technologies. In *Proceedings of the 25th IEEE International Symposium on Rapid System Prototyping (RSP'14).*

[14] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. 2015. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of the 2015 USENIX Anual Technical Conference (ATC'15).*

[15] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, Cheng-Yuan, and Michael Wang. 2014. NVM duet: Unified working memory and persistent store architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14).*

[16] Youyou Lu, Jiwu Shu, and Long Sun. 2015. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST'15).*

[17] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Nathan Binkert, and Parthasarathy Ranganathan. 2013. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the ACM Conference on Timely Results in Operating Systems (TRIOS'13).*

[18] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. 2015. SQLite optimization with phase change memory for mobile applications. *Proc. VLDB Endow. (PVLDB)* 8, 12 (2015), 1454–1465.

[19] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD'16).*

[20] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA'14).* 265–276.

[21] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX conference on File and Storage Technologies (FAST'11).*

[22] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11).*

[23] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. 2014. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the ACM SIGOPS/Eurosys European Conference on Computer Systems (EuroSys'14).*

[24] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A file system for storage class memory. In *Proceedings of the ACM/IEEE SC2011 Conference.*

[25] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, and Khai Leong Yong. 2015. NV-Tree: Reducing consistency const for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15).*

Received February 2017; accepted July 2017