ROART: Range-query Optimized Persistent ART

Shaonan Ma and Kang Chen, *Tsinghua University;* Shimin Chen, *SKL of Computer Architecture, ICT, CAS, and University of Chinese Academy of Sciences;* Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu, *Tsinghua University*

https://www.usenix.org/conference/fast21/presentation/ma

This paper is included in the Proceedings of the 19th USENIX Conference on File and Storage Technologies.
February 23–25, 2021
978-1-939133-20-5

ROART: Range-query Optimized Persistent ART

Shaonan Ma[1], Kang Chen[1]*, Shimin Chen[2,3], Mengxing Liu[1], Jianglang Zhu[1], Hongbo Kang[1], and Yongwei Wu[1]

[1]*Tsinghua University*,† [2]*SKL of Computer Architecture, ICT, CAS*

[3]*University of Chinese Academy of Sciences*

Abstract

With the availability of commercial NVM devices such as Intel Optane DC PMM, it is time to start thinking about applying the existing persistent data structures in practice. This paper considers three practical aspects, which have significant influences on the design of persistent indexes, including **functionality, performance** and **correctness**.

We design a new persistent index, ROART, based on adaptive radix tree (ART), taking all these practical aspects into account. ROART (i) proposes a *leaf compaction* method to reduce pointer chasing for range queries, (ii) minimizes persistence overhead with three optimizations, i.e., *entry compression*, *selective metadata persistence* and *minimally ordered split*, and (iii) designs a fast memory management to prevent memory leaks, and eliminates the long recovery time by proposing an *instant restart* strategy. Evaluations show that ROART outperforms the state-of-the-art radix tree by up to 1.65× and B^+-Trees by 1.17∼8.27× respectively.

1 Introduction

Emerging Non-Volatile Memory (NVM) is attractive because of its byte-addressability, low latency and durability. Many researchers have focused on how to design fast persistent data structures [1–30]. With the announcement of the first generation products (Intel Optane DC PMM [31]), it is time to investigate how to apply the achieved results in practice.

We point out that there are three significant aspects affecting the design of persistent indexes, i.e., functionality, performance, and correctness. Any persistent index designed for practical uses needs to consider these aspects carefully.

1. Functionality: Variable-sized Keys and Range Queries. Variable-sized keys are important in real world systems, such as RDBMS [32–41] and Key-Value Stores [25, 42–47]. Whether an index supports variable-sized keys will have a great impact on the direction of its optimization. Moreover, range queries are used to support inequality comparisons in many real-world applications [32–41]. Therefore, it is desirable that the index structure supports range queries efficiently in addition to point read and write operations. In this paper, we focus on index structures that support both variable-sized keys and range queries.

2. Performance: Persistence Overhead. Persistence overhead plays an essential role in the performance of indexes targeting NVM. To guarantee crash consistency of indexes, once an operation is completed, the modification must be persisted to NVM by cache line flush and memory fence instructions. Due to the design of hardware, NVM writes have lower throughput than reads and poor scalability of bandwidth (because the writes issued by more threads exceed the capability of underlying buffers [48]). Moreover, persistence operations incur much larger (e.g., at least by 2.4×) overhead than normal writes.

3. Correctness: Anomaly Resolution and Memory Safety. First, persistent indexes may suffer from anomalies [26], such as lost update and dirty read, if they provide no protection to concurrent operations. These anomalies will cause the effect of successful operations to disappear after system crash and restart. Second, memory allocations in NVM need to deal with crash consistency, which is not a problem in DRAM. Memory leaks may happen after a crash due to (i) inconsistent memory allocation metadata, and/or (ii) lazy GC (garbage collection) used in the design of non-blocking data structures.

In order to support range queries, our work mainly focuses on tree-based indexes rather than hash tables. According to our experiments (§2.1), B^+-Trees may not be the best performing index for variable-sized keys. Therefore, we choose the radix tree as our basis, which naturally supports variable-sized keys. To the best of our knowledge, no recent persistent radix tree has fully taken the above aspects into account. WORT [19] is write-optimized but only runs in a single thread. P-ART [24] is a concurrent persistent radix tree converted

*Corresponding author: Kang Chen (chenkang@tsinghua.edu.cn)

†Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China

from the volatile ART [49, 50] index based on the principles of RECIPE. There is little optimization on range queries and persistence overhead. Moreover, neither of the two radix trees considers memory safety, which may lead to memory leaks.

We propose a new index structure called ROART (**R**ange-query **O**ptimized **A**daptive **R**adix **T**ree), considering all the above factors. To improve range queries, we propose *leaf compaction* (LC) that delays the leaf split and compacts multiple leaf nodes into a leaf array. The benefits of this technique are threefold. First, it reduces pointer chasing during range queries. Second, it can decrease the number of complex split operations. Finally, it tends to lower the height of the tree, which is beneficial to all index operations.

To reduce persistence overhead, we propose three optimizations in ROART: (i) *Entry compression* (EC) that combines the key and the child pointer in an 8-byte entry; (ii) *Selective Metadata Persistence* (SMP) to reduce the amount of metadata to persist; and (iii) *minimally ordered split* (MO) that relaxes the order of steps in a split operation to reduce the number of `sfence` instructions. Previously mentioned LC also helps here because it delays leaf node split and reduces its persistence overhead.

For correctness, we protect ROART against anomalies by using `non-temporal store` [24, 48] techniques. For memory safety, there have been several previous proposals. Logging-based allocators [51–53] suffer from heavy persistence overhead for allocation/deallocation operations. Post-crash garbage collection techniques [22, 54–56] reduce the persistence overhead, but introduce long recovery time. We propose a new technique, called *instant restart*, with concurrent post-crash garbage collection. While performing the background GC during recovery, indexes can handle foreground requests concurrently, i.e., restarting services instantly. We integrate this technique in our new memory allocator, called DCMM (**D**elayed **C**heck **M**emory **M**anagement).

In summary, this paper makes the following contributions:

(1) We present an in-depth analysis on the three practical aspects of persistent indexes to understand the impact of different design choices (§2).

(2) We propose ROART that addresses the three design aspects (§3). For functionality, we choose ART as basis to naturally support variable-sized keys and propose a *leaf compaction* method to optimize range queries. For performance, we propose three techniques to reduce persistence overhead, i.e., *entry compression*, *selective metadata persistence* and *minimally ordered split*. For correctness, we carefully protect ROART against anomalies, and design DCMM with *instant restart* to support memory safety without long recovery time.

(3) We perform extensive experiments to compare ROART with state-of-the-art tree-based indexes (§4), including P-ART [24], PMwCAS-ART (implemented with PMwCAS [23]), FAST&FAIR [5], SkipList [22] and BzTree [6]. ROART outperforms the existing solutions by 1.15∼8.27× under YCSB workloads.

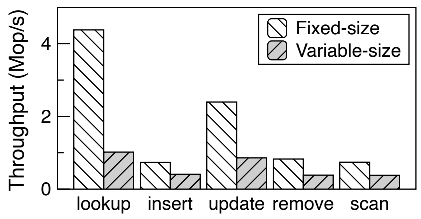


Figure 1: Performance degradation of storing variable-size data in FAST&FAIR.

2 Practical Considerations

We discuss the three aspects, i.e. functionality, persistence overhead, and correctness, in detail in this section.

2.1 Functionality

2.1.1 Variable-sized Keys

Indexes supporting variable-sized keys can be applicable to a wider range of applications, including database systems [32–41]. However, such support may come with a price.

A number of persistent B^+-Tree indexes maintain fixed-sized (8-byte key and 8-byte value) entries in the array of nodes such as NVTree [3], wB$^+$-Tree [2], FPTree [4], RN-Tree [7] and LB$^+$-Tree [9], and entries are appended to the arrays. FAST&FAIR [5], which also supports only fixed 8-byte keys, reduces the number of `clwbs` if multiple keys and values are in the same cache line. These B^+-Trees have excellent cache locality and high traversal performance based on optimizing 8-byte keys only.

A straightforward way to adapt the indexes with 8-byte keys to support variable-sized keys is to allocate extra data areas and store the addresses of the keys in the indexes. However, this incurs pointer chasing overhead. We use FAST&FAIR, a state-of-the-art persistent B^+-Tree, as an example to reveal the performance degradation using such a method. In Figure 1, we evaluate the performance of five operations (lookup/insert/update/remove/scan) with four threads using modified FAST&FAIR. We use the above method to support variable-sized keys, and use an appropriate NVM allocator (with post-crash GC (§2.3.2)) to eliminate the persistence overhead during allocation, so that we can focus on the performance difference between fixed-sized and variable-sized keys. We find the degradations of the five operations are about 3.9/1.8/2.79/2.15/1.94× respectively. The main performance difference comes from pointer chasing and string comparison during traversal. To persist extra data areas, operations like insert/update introduce more persistence.

BzTree [6] employs a different approach, slotted pages, for variable-sized keys. Based on this approach, fixed-sized metadata grows downward into the node, and variable-sized keys and values grow upward. Such an approach can reduce pointer chasing during traversal, but introduce the cost of additional metadata [8]. The performance of BzTree is in Figure 14, its lookups are fast but writes are slow.

Indexes based on radix tree [49] have better performance (Figure 14) in supporting variable-sized keys than B^+-Tree due to less comparison in traversal. However, they also have their own shortcomings, such as inefficiency on range queries.

2.1.2 Range Queries

Range query is an important feature in RDBMS [32–41] and Key-Value Stores [25, 42–47]. This paper focuses on tree-based indexes which naturally support range queries. B^+-Trees support efficient range queries because multiple keys are stored in one leaf node, and scan in leaf nodes causes no pointer chasing. In other tree structures, such as radix trees and binary search trees, one node can only store one key, and keys can be stored in leaf nodes but also in non-leaf nodes. Range queries on these trees have to traverse different levels of the trees, and chase more pointers. As the performance gap between sequential read and random read is larger in NVM than that in DRAM [48], more random accesses deteriorate the range query performance in NVM.

However, B^+-Trees may not be the best choice for indexes when both variable-sized keys and range queries are required. In Figure 1, the scan throughput of B^+-Trees decreases by 48.5% when variable-sized keys are used. In §3, we optimize the adaptive radix tree (ART) for range queries.

2.2 Persistence Overhead

Explicit persistence is required to guarantee crash consistency for indexes in NVM. However, cache line flush and memory fence are costly compared to other instructions. Due to the poor scalability of NVM writes [48], reducing the persistence overhead is crucial for improving performance.

There are several general methods to convert a volatile index to its non-volatile counterpart in NVM. PMwCAS [23] records the metadata of each CAS (`Compare-and-Swap`) into a descriptor to ensure multiple CASs can execute atomically. RECIPE [24] adds a persistent instruction (flush and fence) after each store to ensure persistence. Unfortunately, without any optimization to reduce persistence operations, such generality comes with high overhead [8].

Other works propose special optimizations to eschew heavy persistence. wB^+-Tree [2], NVTree [3], FPTree [4], and LB^+-Tree [9] abandon the order of keys in leaf nodes to reduce writes for insertions and deletions. FAST&FAIR [5] and LB^+-Tree [9] reduce persistence operations by making data share the same cache line as much as possible. RNTree [7] uses HTM [57] to increase the granularity of atomic writes to reduce the number of persistence operations. From the above, we see that it is important to exploit the properties of target data structures. In this paper, we optimize persistence in ROART by exploiting inherent properties of ART (§3).

Table 1: States of three steps in an insert operation.

next pointer	Step (i)	Step (ii)	Step (iii)
volatile state	old	new	new
persistent state	old	old	new

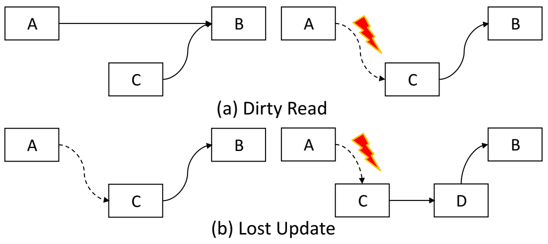


Figure 2: Two anomalies in unmodified lock-free linked list. The dotted line indicates that it has not been persisted.

2.3 Correctness

2.3.1 Anomaly Resolution

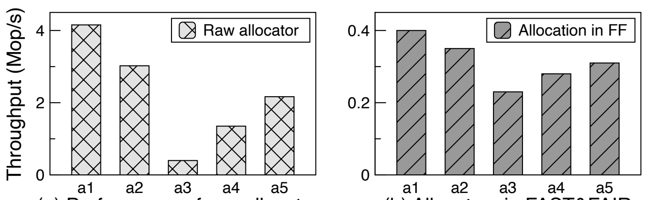
When designing lock-free non-volatile data structures, two anomalies (`Dirty Read` and `Lost Update`), are prone to occur [26]. The main reason is that threads may access data yet to be persisted, which are lost after crash and restart.

We use the lock-free linked list [58] as an example to describe the two anomalies. The insert operation has three steps. Step (i) creates a new node, sets its next pointer to point to the successor, and then persists the new node. Step (ii) updates the next pointer of the predecessor to point to the new node. Step (iii) persists the next pointer of the predecessor. The states of the predecessor's next pointer are shown in Table 1. Note that there is an inconsistency between volatile and persistent states in step (ii). Without extra protection mechanisms, two anomalies can happen, as illustrated in Figure 2.

Dirty Read. In Figure 2(a), an insert operation inserts a new node C between A and B. Suppose, the operation finishes step (ii) but has not executed step (iii) yet. At this moment, a concurrent read operation visits C. If the system crashes at this point, C is lost after restart and the read operation has read the uncommitted dirty data.

Lost Update. In Figure 2(b), two insert operations want to insert two adjacent nodes C and D between A and B. The successful result should be $A \rightarrow C \rightarrow D \rightarrow B$. Suppose that a thread completes step (i) and (ii) for inserting C. Then another thread inserts D between C and B, and completes all three steps. If the system crashes before the insert of C completes step (iii), we can only recover $A \rightarrow B$ from NVM, but lose the completed insert operation of D.

These anomalies can be fixed in various ways. For lock-free designs, `link-and-persist` [22] and `PMwCAS` [23] can be used, e.g., BzTree stays away from anomalies by `PMwCAS`, SkipList can ensure correctness by `link-and-persist`. For lock-based designs, implementations can replace `temporal store` with `non-temporal store` [24], such as P-ART [24].



(a) Performance of raw allocators (b) Allocators in FAST&FAIR

Figure 3: Comparison of different allocators.

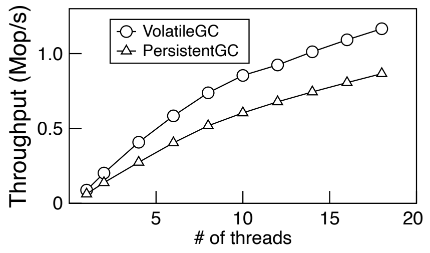


Figure 4: Volatile GC vs. Persistent GC.

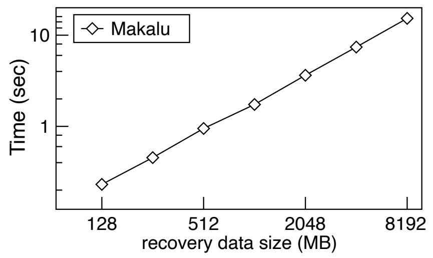


Figure 5: Recovery time of Makalu.

Table 2: Analysis of recent works[1].

Name	Functionality		Correctness		
	Variable	Range Query	Anomaly	Allocation	GC
NVTree	●	✓	▼	▲	▼
wB$^+$-Tree	●	✓	▼	▲	▼
FPTree	●	✓	✓	▲	▼
FAST&FAIR	●	✓	✓	✖	✖
RNTree	●	✓	✓	✖	✖
WORT	✓	●	▼	✖	▼
BzTree	✓	✓	✓	✓	✓
P-ART	✓	●	✓	✖	✖
LB$^+$-Tree	●	✓	✓	▲	▼

NOTE: ▲: not open-sourced ▼: no need ●: not optimized
✓: support ✖: poor support

2.3.2 Memory Safety

It is important to guarantee crash consistency for NVM memory management. Inconsistent metadata for allocation/deallocation operations and/or lazy garbage collection can lead to memory leaks.

NVM Allocation. Many existing studies employ the simplistic solution that uses volatile allocators, such as `malloc`, `libvmmalloc` [59], for persistent memory. An allocation typically consists of three steps: (i) allocate a free NVM block, (ii) modify the allocator's metadata, and (iii) return the allocated block to the application. However, these volatile allocators pay no attention to crash consistency of allocators' metadata. Once the system crashes during the allocation process, the metadata is likely to be inconsistent, making part of the NVM memory unreachable. Moreover, volatile allocators can result in inaccurate performance evaluation for applications based on them [48], because they overlook the expensive persistent instructions. Therefore, using volatile allocators directly for NVM is not appropriate.

Correct NVM allocators are divided into two categories, logging-based allocator [51–53] and post-crash GC [22, 54–56]. The former uses logging to ensure the atomicity of operations, introducing additional persistence overhead. The latter reclaims garbage data by scanning all memory space during recovery. It reduces persistence overhead during allocation/deallocation operations, but suffers long recovery time as the amount of data increases.

Allocation Performance. We evaluate five commonly used (volatile and persistent) NVM allocators as follows:

a1: malloc, the standard volatile allocator for DRAM.

a2: libvmmalloc, volatile allocator based on `jemalloc`.

a3: PMDK [51], logging-based persistent allocator.

a4: nvm_malloc [52], logging-based persistent allocator.

a5: Makalu [54], persistent allocator with post-crash GC.

Figure 3(a) shows the raw performance of the allocators. The test continuously allocates 64-byte chunks, then writes and persists them in a single thread. Makalu is 50% and 28% slower than malloc and libvmmalloc, respectively. PMDK and nvm_malloc are 81% and 38% slower than Makalu, respectively. Performance of FAST&FAIR using different allocators is shown in Figure 3(b). We see that the gaps between different cases are narrowed due to the tree's traversal overhead. Makalu is 22% and 12% slower than malloc and libvmmalloc, respectively. PMDK and nvm_malloc are 25% and 10% slower than Makalu, respectively.

Garbage Collection. Many indexes support non-blocking lookups [60–62] to improve the read performance. Lazy GC mechanism is necessary in such implementations. An item to delete is firstly labeled as *logically* deleted before it is *physically* deleted for avoiding dangerous concurrent accesses from other threads. After a grace period, GC threads sweep and collect the *logically* deleted items.

For example, epoch-based GC [63] is a commonly used strategy for lazy GC [60, 61, 64]. However, a volatile epoch-based GC implementation may incur memory leaks in NVM because it does not persist the metadata of labeled garbage data. After restart, these garbage data will be unreachable.

A naïve way to address this problem is to persist the metadata of labeled garbage data for every metadata modification. We compare the volatile and naïve persistent epoch-based GC in FAST&FAIR, which supports lock-free lookup operations and needs a GC mechanism. As shown in Figure 4, we see that the performance of persistent epoch-based GC is 25.7% worse than the volatile GC.

[1] *not open-sourced* means that we are not sure what it uses. *no need* means that it does not need to consider this factor.

Table 3: Optimizations on practical aspects in ROART.

	ROART Design
Functionality	*Leaf Compaction* (§3.2) to optimize range queries.
Performance	*Entry Compression* (§3.3) to use only one persistent instruction for structural information.
	Selective Metadata Persistence (§3.3.1) to reduce the amount of persisted metadata.
	Minimally Ordered Split (§3.3.2) to reduce `sfence` instructions in internal node split.
Correctness	`non-temproal store` to resolve anomaly (§3.4.1).
	DCMM (§3.4.2) to prevent memory leaks with minimal persistence during allocation. *Instant Restart* to eliminate the long recovery time for DCMM.

Post-crash GC [22, 54–56] reduces the persistence overhead during normal operations. We test the recovery time of Makalu [54] with post-crash GC in Figure 5, and find that the recovery time increases linearly as the amount of data increases.

We would like to design a GC solution that neither incurs the persistence overhead in the naïve epoch-based GC nor suffers long recovery time of the post-crash GC.

In summary, Table 2 compares the functionality and correctness features of recent studies. We find that most studies do not consider all these aspects (BzTree takes these into account, but it suffers heavy persistence overhead [8]). Thus, there are still a lot of opportunities for improvement.

3 Design of ROART

Based on the discussion and analysis above, we propose a persistent index, ROART, which takes the three aspects into account. Table 3 summarizes the distinct features of ROART.

3.1 Radix Tree and its Persistent Variants

A radix tree is a search tree in which each node represents a chunk of bits in the key. The key in a leaf node equals to the string constructed along the path, starting from the root to the corresponding leaf node. Suppose one node stores s bits of a key. The node has at most $r = 2^s$ children. r is called the *radix* of the tree. ART [49, 50] is a space-efficient radix tree. Its *radix* is 256 and each node represents a 1-byte character (8-bit, $r = 256 = 2^s$, $s = 8$) of the key. We will discuss its node types in §3.3.

Path Compression. The height of the radix tree can be reduced by path compression [49], as illustrated in Figure 6. A node with only one child is merged into its child, and the character it represents is merged into the prefix of its child.

Node Split. With path compression, node splits may happen during insertions. Node splits are divided into two categories, i.e., internal node split and leaf node split, as shown in Figure 7. An internal node split occurs when a new insertion (e.g., $L3$) mismatches the prefix of an internal node (*old*). The node *new* is created and inserted into the tree, which points to $L3$

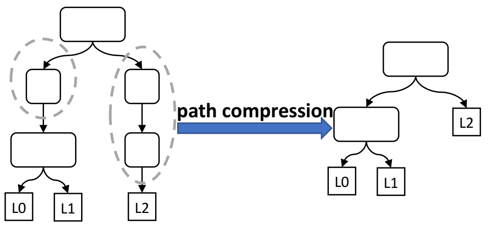


Figure 6: Path compression in radix tree.

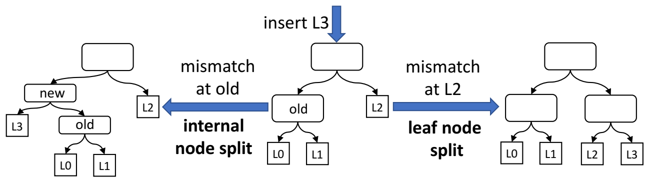


Figure 7: Two ways of node split.

and *old* as its children. The prefix of *old* will be updated. A leaf node split occurs when a new insertion ($L3$) mismatches the key in a leaf node ($L2$). A new node pointing to both $L2$ and $L3$ will be created and inserted into the tree.

Persistent Variants. Because of ART's efficiency, most persistent radix trees are based on ART. The ART implementation in WORT [19] supports only single thread, and uses a separate slot array and a bitmap in its node to help locate entries. However, this incurs extra persistence overhead. RECIPE [24] proposes a method to convert any volatile data structure to its persistent counterpart. Based on this method, P-ART is directly converted from the volatile ART-ROWEX [50]. However, this leaves many opportunities for persistence optimizations. Neither WORT nor P-ART has optimized range queries.

3.2 Our Solution: ROART Structure

Compared to B$^+$-Trees, the radix tree performs poorly in range queries because each leaf node stores only a pair of key and value, and leaf nodes can be on many different levels in the tree. A range scan has to visit a large number of non-leaf nodes on different levels in addition to leaf nodes, incurring significant overhead. We propose *leaf compaction* (LC) to compact the pointers of leaf nodes into a *leaf array* in the radix tree. A leaf array can contain up to m leaf pointers. If a subtree of the radix tree has less than or equal to m leaf nodes, the subtree is compacted into a leaf array. (We set $m = 64$ in our implementation.) For simplicity of presentation, the figures in this subsection use $m = 4$.

Figure 8(a) and 8(b) show the structural differences between ART and ROART with the same leaf nodes. We see that the subtrees rooted at B and D are compacted into leaf array F and G, respectively. For range queries, *leaf compaction* can effectively reduce the number of pointer chasing in the different levels of the tree. For instance, to run a range query covering $L0 - L5$, ART dereferences 15 pointers $(A - B - C - L0 - L1 - C - B - L2 - B - A - D - L3 - E -$

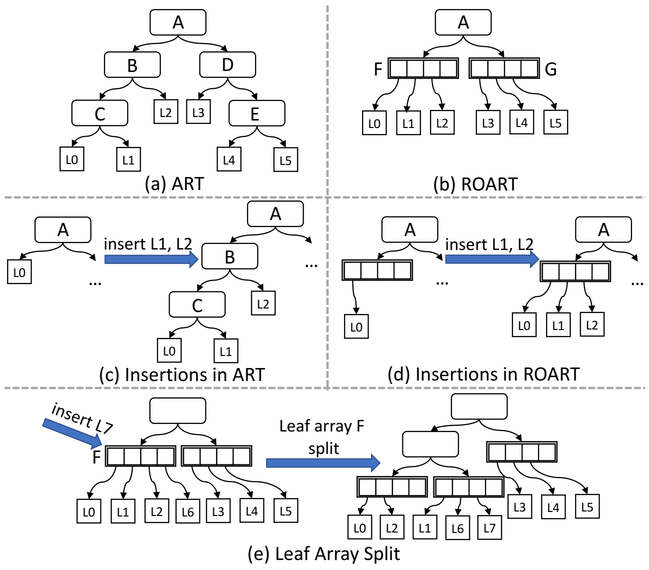


Figure 8: Leaf compaction in ROART.

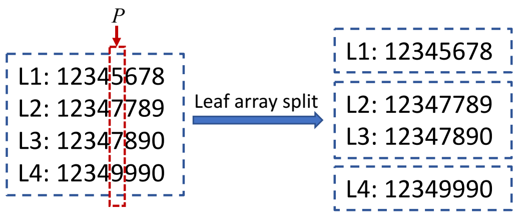


Figure 9: An example of leaf array split.

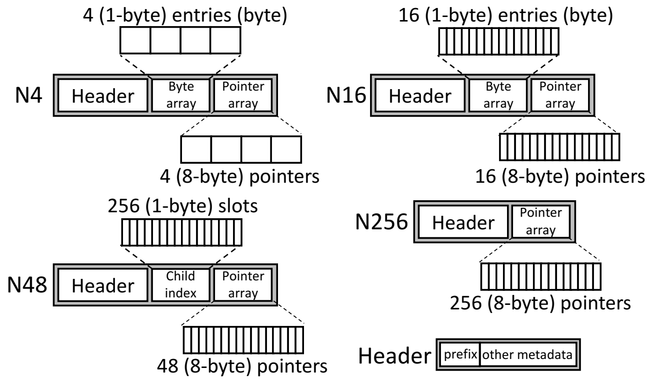


Figure 10: Node types in ART.

$L4-L5$), while ROART requires only 11 pointer dereferences ($A-F-L0-L1-L2-F-A-G-L3-L4-L5$).

We modify the index operations to support leaf compaction. For lookup, a reader searches the tree as before until it reaches a leaf array. It has to check each leaf node that the leaf array points to, which can be costly. To minimize this effect, we embed a 16-bit fingerprint (hash value) [17, 18] of each leaf key into the pointer in the leaf array (current architectures support only 48-bit addresses), represented as | fingerprint: 16-bit | address: 48-bit |. In this way, the reader compares the fingerprint of the search key with the fingerprints in the leaf array to filter out most unnecessary cases. The probability of false positives is low (e.g., < 0.001 for $m = 64$).

For insert, when it reaches a leaf array, a writer checks to see if the key already exists. If not, the writer chooses an empty slot to insert, as shown in Figure 8(d). The complex case is when the leaf array is full and the leaf array splits, as shown in Figure 8(e). Here, the writer wants to insert a new leaf $L7$ into leaf array F, which is already full. Note that all keys in F correspond to a subtree in the original radix tree, and therefore share a common prefix. To split, we need to find the first byte position, denoted as P, where the keys diverge. We call the P-th byte as the identifying byte. We divide the keys into subsets based on their identifying bytes. We build a leaf array for each subset, and create a new internal node, which contains each identifying byte and the pointer to the associated leaf array. For example, in Figure 9, a leaf array with four leaf pointers is to split into three new leaf arrays. The keys diverge at the 5th byte, i.e. $P = 5$. The keys are divided, and then new leaf arrays are created. Note that the cost of a leaf array split is high, but fortunately, this is a rare operation.

For update and delete, the procedure is similar to lookup. After the matching key is found, the corresponding modification or deletion is performed.

For range query, the only difference with ART is that keys in a leaf array are not sorted, but keys are ordered between leaf arrays. Thus, we only need to check/sort the begin and the last leaf arrays to ensure that the return values are within the requested range. If values need to be fully sorted before returned, it will bring about 8.9% performance degradation with some optimizations, e.g., by skipping the prefix of keys and comparing only the different parts of keys.

Interestingly, *leaf compaction* can improve not only range queries but also traversals and insertions. Traversal is an essential step in all operations (lookup/insert/update/delete/scan). *Leaf compaction* tends to reduce the root-to-leaf path lengths, as can be clearly seen in Figure 8(a) and (b). Shorter path lengths are beneficial to traversals. Moreover, when a new insertion mismatches the key in a leaf node, ART incurs a leaf node split (Figure 7) while ROART simply inserts the new leaf in the leaf array, reducing the number of persistent instructions. (Please see Table 4 for detailed counts.)

In summary, *leaf compaction* has several benefits: (i) decreasing the number of pointer chasing for range queries, (ii) reducing root-to-leaf path lengths and traversal overhead, (iii) reducing persistence overhead of insertions. We will evaluate the benefits of this structure in §4.

3.3 Reducing Persistence Overhead

Figure 11 shows the node structures of ROART, which are based on the design of ART. Figure 10 shows the four types of nodes in ART that can store up to 4, 16, 48, and 256 entries, respectively. Each entry contains a `byte` and a child pointer. The `byte` is equal to the character represented by the corre-

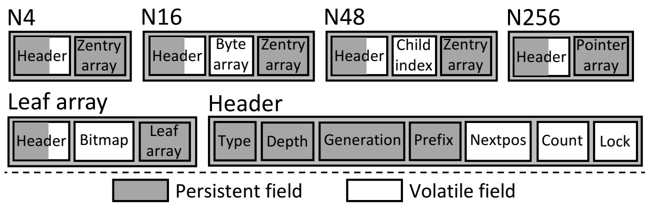


Figure 11: Node structures in ROART.

sponding child node. In N4 and N16, `bytes` and pointers are stored in `Byte array` and `Pointer array`, respectively. In N48, `Child index` has 256 slots so that `bytes` can be used as an index to search in `Child index` to find the location of the corresponding child pointer, because the range of a `byte` is 0-255. N256 directly has an array with 256 pointers. A node expands to a larger node type when it is full and a new entry is to be inserted, and shrinks to a smaller node type when the number of entries is below a threshold.

In ROART, we propose *entry compression* (EC) to pack the key byte into the pointer (where 48-bit are used) in N4, N16, and N48. The resulting entry | empty: 8-bit | key: 8-bit | pointer: 48-bit | is called `Zentry`, as shown in Figure 11. An invalid `Zentry` (for a deleted or unused slot) is set to 0. Since a `Zentry` is 8-byte and can be updated atomically, compared to ART, EC reduces one persistent instruction for persisting each entry in ROART.

3.3.1 Selective Metadata Persistence

We observe that not all metadata need to be persisted for correctness. For example, `Nextpos` and `Count` in the header indicate the next empty entry slot and the number of used slots. The `Bitmap` in the leaf array shows which leaf array entry is in use. They can all be computed by scanning the Zentry or the pointer array, where empty slots are set to 0. Moreover, the `Byte array` in N16 is used to accelerate search with SIMD instructions. It can be rebuilt by retrieving the embedded key byte from each Zentry. The `Child index` in N48 can be restored in the same way. Finally, `Lock` is used for concurrency control and can be cleared upon crash recovery.

Based on this observation, we propose *selective metadata persistence* (SMP) to selectively persist a subset of the metadata and recompute the rest of the metadata after recovery. As shown in Figure 11, volatile metadata are highlighted with white background. Fields with grey background are persisted.

Traditional recovery needs to suspend processing requests and scan the whole indexes, which incurs long recovery time as the amount of data increases. Inspired by the generation lock in NV-Heaps [65], we implement *selective metadata persistence* using generation numbers to hide the recovery overhead. ROART maintains a global generation number (GGN) in NVM. GGN is increased upon each restart. Each node in ROART has its own persistent node generation number (NGN). When accessed, if NGN equals to GGN, the metadata in the node is up-to-date. Otherwise, the metadata in the node is restored, then GGN is assigned to NGN. Per-node latch for recovery (implemented by using flags in memory and CAS instructions) protects the concurrent access on the same node from multiple threads. In this way, after restart, ROART does not suspend normal operations for recovering the whole lost metadata. Instead, it restores the metadata on demand and at the same time executes normal operations.

Table 4: Persistence analysis. Two values are the numbers of `clwb` and `sfence` respectively (lower is better).

Name	Insert				Split	
	N4	N16	N48	N256	Leaf	Internal
ROART	2, 2	2, 2	2, 2	2, 2	2, 2	4, 2
P-ART	2, 2	3, 3	3, 3	2, 2	3, 3	4, 4
WORT	3, 3	4, 4	3, 3	2, 2	3, 3	4, 4

3.3.2 Minimally Ordered Split

Internal node split is costly as shown in Figure 7. It has four steps: (i) allocating a new leaf $L3$, (ii) allocating an internal node, marked as *new*, with two children ($L3$ and node *old*), (iii) changing the pointer (from *old* to *new*) of parent node, (iv) updating the prefix of *old* (not shown in the figure). Without optimizations, the four steps need four `sfence` instructions.

We observe that the order of these four steps can be relaxed. Step (i) and step (ii) are not visible to other threads, we can use only one `sfence` after initializing the two nodes. Step (iii) and (iv) cannot execute atomically. Under concurrent execution, readers may see the incomplete split with inconsistent prefixes. Note that the depth of a node (including the prefix) stays constant. This property can be exploited to detect inconsistent prefixes, as is also used in other work [19, 50]. Once such inconsistency is detected, it is easy to repair the inconsistency by recomputing the prefix. Consequently, the order of step (iii) and step (iv) is not important.

ROART performs the internal split as follows. It performs step (i), (ii) and (iv), flushes the modified cache lines, then calls a single `sfence`. After that, it performs step (iii), flushes the modified cache line, and calls a second `sfence`. In this way, ROART reduces the numbrer of `sfence` instructions of an internal split from four to two.

3.3.3 Persistence Analysis

Table 4 compares the number of persistence instructions, `clwb` and `sfence`, for insert and split operations in ROART, P-ART [24], and WORT [19]. ROART incurs the smallest number of persistence instructions among the three indexes.

3.4 Making ROART Correct

3.4.1 Anomaly Resolution in ROART

ROART employs a concurrency control strategy similar to ART-ROWEX [50], i.e., lock-free read and lock-based write.

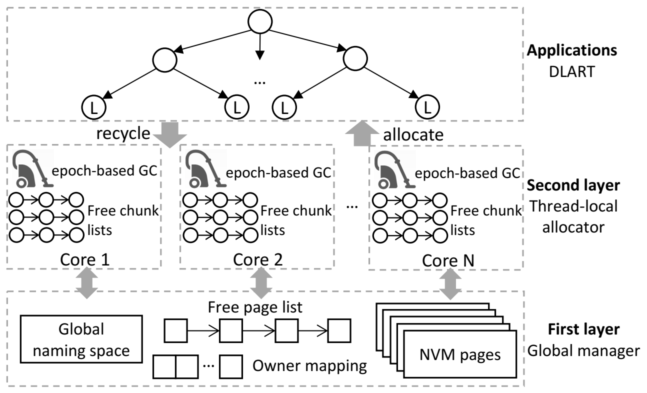


Figure 12: DCMM architecture

Readers may see incomplete write operations, resulting from inconsistency between volatile and persistent states. So extra protection is required. §2.3.1 discusses several methods to address this problem. We adopt `non-temporal store` [24] to fix the potential anomalies in ROART.

3.4.2 Delayed Check Memory Management

NVM allocation and GC have a large performance impact on practical indexes (§2.3.2). We propose a new memory management method, called DCMM, which uses post-crash GC to minimize the persistence during allocation/deallocation, and supports *instant restart* to eliminate the waiting time after restart. To reduce the contention in memory allocation for multiple threads, DCMM uses a two-layer architecture, as in Figure 12.

First Layer. This layer is a global memory manager that manages the entire NVM area at the granularity of pages. The page size is adjustable, and defaults to 128MB [53, 56]. The global memory manager maintains a global naming space. It contains the roots of indexes and an `offset` field indicating the offset of the last allocated page. These are persistent fields. There are also two volatile fields, i.e., `owner_mapping` and `free_page_list`. The former keeps a map between each page and its owner thread, and the latter implements a volatile lock-free linked list for recycled free pages. A thread requests a new page by searching `free_page_list`. If `free_page_list` is empty, it uses an atomic `fetch-and-add` to obtain the `offset` and increase the `offset` by the page size, then persists the `offset`.

Second Layer. For each application thread, a thread-local allocator performs allocation using multiple block classes with different sizes. Each block class is maintained by a volatile linked-list, called `free_chunk_list`. A thread requests a page from the first layer and divides it into `free_chunk_lists`, like in the buddy system [66].

Garbage Collection. DCMM implements post-crash epoch-based GC for supporting lock-free reads and lazy deletions.

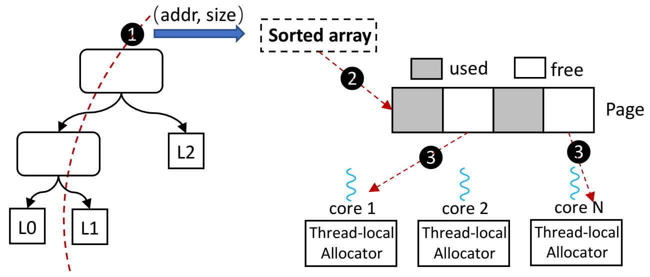


Figure 13: The procedure of recovery.

We implement a decentralized version [60] of epoch-based GC for better scalability.

Persistence Overhead. As discussed in §2.3.2, the allocator based on post-crash GC does not need to persist any metadata during normal operations. Therefore, persistence overhead only occurs in the first layer, for persisting `offset`. This overhead is amortized by multiple memory allocations in the second layer. Most allocations do not invoke the first layer.

Recovery Processing. Upon recovery, `owner_mapping` can be simply reset by mapping each page in the range [0, `offset`) to application threads in the round-robin fashion. Other volatile information can be restored by the recovery process in three steps (Figure 13) : (i) Recovery threads traverse all NVM areas used by the application (i.e., starting from the persistent index root pointers and traversing the trees), and collect all used chunks with their description tuples (address, size). (ii) Free chunks can be calculated based on the used chunks in each page. (iii) Free chunks are put in the `free_chunk_lists` and free pages are put in the `free_page_list`.

Instant Restart. The recovery process as described in the above can take a long time as the amount of data increases (Figure 5). In order to reduce the waiting time after restart, we propose *instant restart* for DCMM. Note that `offset` in the first layer is persisted. Therefore, after restart, we can immediately allocate new pages after `offset` without waiting for other metadata recovery to complete. (If `offset` exceeds the current NVM file limit, a new NVM file will be created and opened for allocation [53, 56].) Hence, we can immediately allow front-end operations and provide memory allocation service instantly after restart, while the background recovery threads run in parallel. In this way, DCMM avoids the front-end from waiting long time for the recovery to complete.

Multi-threading Optimization. The background recovery process can be accelerated by multi-threading. Consider the three steps in recovery processing. Step (i) can be parallelized based on the data structures. For example, multiple threads can be used to traverse different subtrees of ROART. The (address, size) pairs produced during traversal can be put into different sorted arrays based on address ranges. Then, Step (ii) and (iii) can examine different sorted arrays and collect free chunks in parallel.

4 Evaluation

Our evaluations consist of four parts to reflect the performance improvements of each proposed design.

1. Overall Performance Comparison. We choose several tree-based data structures in experiments. For a fair comparison, some modifications are necessary such as adding DCMM to some indexes, and implement missing functions.

2. Detailed Test of Each Design. Several aspects are evaluated: (i) performance improvement by each optimization, (ii) range query (scan) performance with different numbers of keys, (iii) fixed-sized (8-byte) keys performance, (iv) skew tests, (v) latency tests, (vi) space consumption, and (vii) recovery and instant restart.

3. NVM Allocators. We evaluate DCMM with several open-sourced persistent allocators, e.g., PMDK [51], nvm_malloc [52] and Makalu [54].

4. Real-world System Evaluation. We incorporate ROART into a real-world system: Memcached [67]. The core index in Memcached is a volatile hash index and our modification enables Memcached to support persistent storage.

4.1 Evaluation Setup

All evaluations use a Dell PowerEdge R740 server with four Intel(R) Xeon(R) Gold 5220 processors supporting `clwb`, 6×128GB Optane DC PMM per socket. The processor has 32KB L1-cache, 1MB L2-cache, and 25MB L3-cache. The persistent memory is managed by a DAX file system [68] and mapped to a pre-defined address. We choose five other tree-based indexes to compare the performance with ROART.

P-ART. P-ART [24] is a persistent counterpart of ART-ROWEX [50]. For a fair comparison, we use DCMM in P-ART, and implement its missing functions (e.g., update operations, and *selective metadata persistence* for metadata).

PMwCAS-ART. PMwCAS-ART is a baseline we implement by using PMwCAS [23]. PMwCAS allows atomically modifying multiple 8-byte words in NVM and uses PMDK to guarantee the memory safety. We leverage the persistent primitives it provides to modify ART-ROWEX.

FAST&FAIR-DCMM. FAST&FAIR [5] is a state-of-the-art persistent B$^+$-Tree (§2). We modify it to support variable-sized keys and use DCMM to manage NVM.

SkipList-DCMM. We modify the open-source lock-free SkipList [22] to support variable-sized keys and use DCMM.

BzTree. BzTree [6] is a lock-free persistent B$^+$-Tree based on PMwCAS. It can naturally support variable-sized keys by using slotted pages.

4.2 Overall Performance

To evaluate the overall performance, we test micro-benchmarks with 4 threads and YCSB benchmark. For the micro-benchmarks, keys are randomly generated with sizes

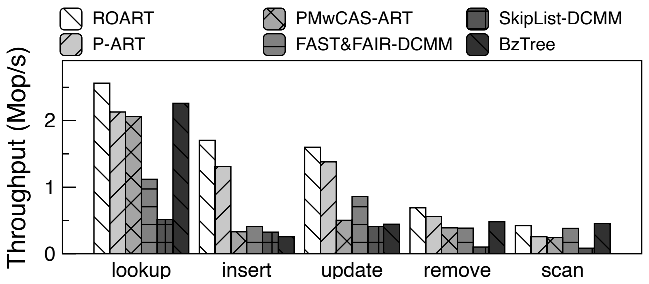


Figure 14: Microbench of six indexes under 4 threads.

between 4 to 128 bytes, and values are fixed as 8 bytes. An 8-byte value can represent an indirect pointer, commonly used in DBMS [32,33]. Each test firstly warms up using 30 million KVs [5, 7, 24], which exceeds the size of the L3-cache and reflects the performance of NVM. After warming up, each test runs 20 seconds for different workloads and reports the average throughput.

Micro-benchmarks contain the operations of lookup, insert, update, remove and scan. The results are in Figure 14. The lookup performance of ROART is 2.562 Mop/s which is faster than P-ART (2.129 Mop/s) and PMwCAS-ART (2.06 Mop/s). The main improvement comes from *leaf compaction* which lowers the height of the tree and benefits traversal. BzTree is fast because its slotted-page node layout has good cache locality for supporting variable-sized KVs and binary search. ROART is 2.29× and 4.98× faster than FAST&FAIR and SkipList respectively, because FAST&FAIR cannot use binary search inside its nodes and SkipList suffers from poor cache locality.

The insert performance of ROART is significantly better (1.704 Mop/s) than all other five indexes (1.3/5.16/4.15/5.24/6.65×). There are several reasons for the improvement. (i) DCMM has higher allocation performance than PMwCAS (PMDK). (ii) Less persistent related instructions (`clwb` and `fence`) in ROART (§3). PMwCAS suffers from more persistent instructions, P-ART makes no optimization, and FAST&FAIR also causes multiple persistent instructions once the entry moving exceeds one cache line. (iii) Compared with B$^+$-Tree (FAST&FAIR, BzTree), no rebalance operations are required in ROART.

For the update operation, ROART can achieve 1.6 Mop/s throughput and outperforms the others up to 1.16/3.18/1.86/3.9/3.61×. The major performance differences are similar to lookup operation. For the remove operation, SkipList performs very poor, and others are similar. The main reason is that SkipList has a complicated remove operation and suffers from many retries. For the scan operation, with *leaf compaction*, the performance of ROART can outperform P-ART up to 1.65× and is close to FAST&FAIR/BzTree.

We use YCSB [69] benchmark to generate five workloads, which are (a) write-intensive (50% lookup and 50% insert), (b) read-intensive (95% lookup and 5% insert), (c) read-only, (d) insert-only, and (e) scan-insert (95% scan and 5% insert).

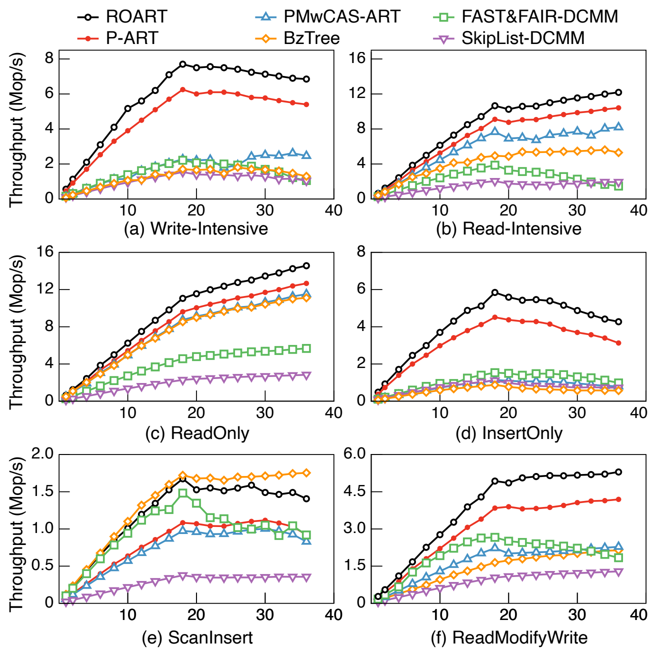


Figure 15: Performance of YCSB.

The results of five workloads are shown in Figure 15.

In workload (a), ROART outperforms P-ART up to 1.27× and other four indexes up to 2.78∼6.57× using 36 threads. The main performance gain is from its less traversal and persistence. In workload (b), ROART outperforms the other five indexes by 1.17∼8.27× using 36 threads. In workload (c), the performance of all indexes is scaled, but ROART can still outperform others by 1.15∼5.13×. In workload (d), due to the influence of NUMA, performance of all indexes begins to decline after more than 18 threads. ROART decreases 27% and P-ART decreases 30% from 18 threads to 36 threads. In workload (e), ROART can perform 1.52× and 1.53× better than P-ART and FAST&FAIR. It is only 3% and 20% slower than lock-free BzTree in the cases of 18 and 36 threads, because BzTree stores variable-sized keys and values in the nodes, instead of extra data areas, located by metadata in the head of nodes.

4.3 Effects of Each Design

1. Improvement of Each Optimization in ROART. We test the performance improvement of each optimization (§3) in Figure 16. The raw version is the implementation of ROART without the four optimizations (SMP/EC/MO/LC). *Selective metadata persistence* can improve by 13% and 10.1% for insertion and deletion. *Entry compression* can bring about 9.7% and 10.8% improvement for insert and remove operations. *Minimally ordered split* reduces the `fence` instructions for internal node split so that it can improve the insert performance by 4.8%. *Leaf compaction* can lower the height of radix tree and benifit every operation, especially scan. It can bring 17.2%/13.5%/14.7%/8.4%/64.8% improvement for

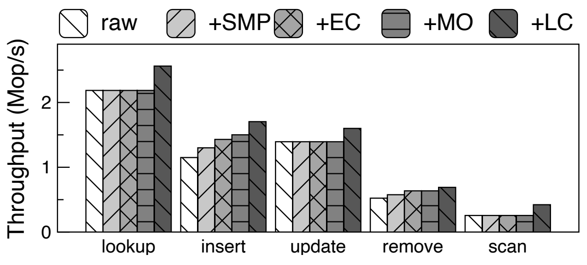


Figure 16: Performance improvement of each optimization. (SMP: Selective Metadata Persistence, EC: Entry Compression, MO: Minimally Ordered split, LC: Leaf Compaction)

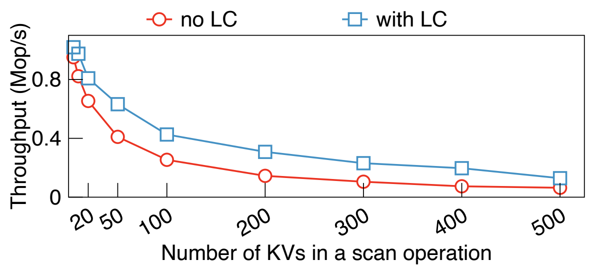


Figure 17: Range queries with different key numbers.

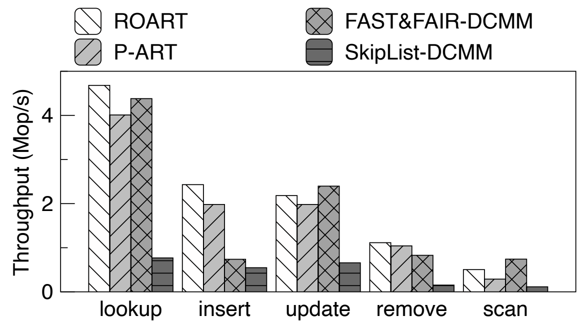


Figure 18: Performance with fixed-sized keys.

the five operations respectively.

2. Range Queries with Different Key Numbers. In Figure 17, we evaluate the range query performance by scan operations with different key numbers, and the necessary parameters of scan are the maximum and minimum keys as well as the number of required keys. The result shows that ROART with LC can outperform the version without LC by 1.07∼2.01×. When the number of keys is less than 50, the improvement brought by LC is not very much, and when the number of keys is more than 100, the performance is improved at least by 1.65×.

3. Performance with Fixed-sized Keys. Many indexes are optimized for fixed-sized KV, such as FAST&FAIR. We test the performance of ROART (without any optimization for fixed-sized keys) while processing 8-byte fixed-sized keys, compared to P-ART, FAST&FAIR and SkipList. The results are shown in Figure 18. SkipList runs slowest because of its poor cache locality. FAST&FAIR outperforms P-ART by up to 1.09/1.21× in lookup and update because fixed-sized

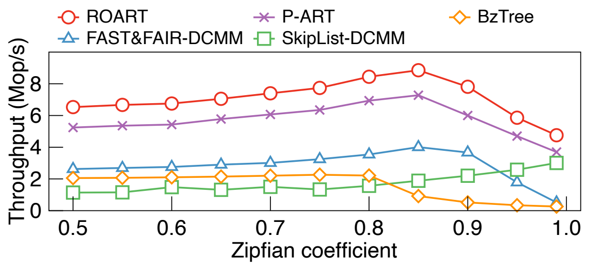


Figure 19: Performance with different zipfian.

Table 5: Latency tests under write-intensive workload (50% lookup and 50% insert) with 16 threads (lower is better).

latency (us)	ROART	P-ART	PMwCAS-ART
avg.	1.2	1.5	3.4
p99	3.5	4.4	8.8
latency (us)	**FAST&FAIR**	**SkipList**	**BzTree**
avg.	3.5	4.4	4.2
p99	11.8	8.9	9.5

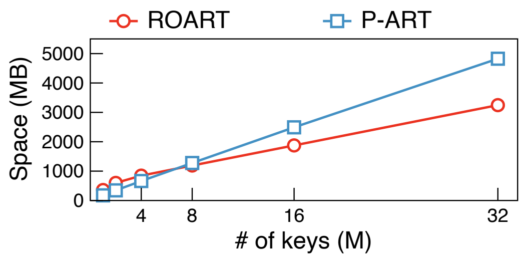


Figure 20: Space consumption of ROART and P-ART

optimized FAST&FAIR has better cache locality. But with the optimization LC, ROART can outperform FAST&FAIR in lookup even with fixed-sized keys, and it is only sightly slower than FAST&FAIR in update. For scan with fixed-sized keys, B^+-Tree is still the better index than radix tree.

4. Skew Tests. Figure 19 shows the experiment under a skewed workload (50% lookup and 50% update with 16 threads). The cache brings more benefits when the coefficient is smaller than 0.85. When larger than 0.85, ROART, P-ART and FAST&FAIR all suffer from the lock contention. The performance of ROART and P-ART drops about 28% and 29.5% from 0.5 to 0.99, while FAST&FAIR drops about 80.3%. Performance of SkipList improves because of its lock-free manner. BzTree drops about 87% because it has a complex structure and heavy persistence overhead [8] although it also has a non-blocking design.

5. Latency Test. Latency numbers of each index are shown in Table 5 under a write-intensive workload (50% lookup and 50% insert) with 16 threads. In average latency, ROART can outperform all other indexes by 20% $\sim$ 73% because of its faster traversal. In p99 latency, ROART can outperform all other indexes by 21% $\sim$ 71%. SkipList and BzTree is lock-free design so that their p99 latencies increase less than other four lock-based indexes. In ROART, we make no extra optimization on tail latency, which is orthogonal to our work.

6. Space Consumption. We introduce leaf arrays in ROART,

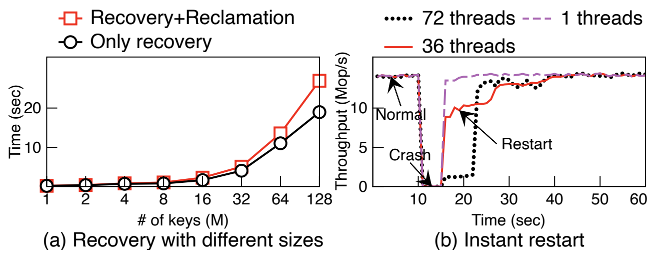


Figure 21: Data structure recovery and instant restart.

which does not exist in ART. In our implementation, the size of leaf array is predefined, which may cause the waste of space. So we evaluate the space consumption of ROART and P-ART to show the impact of leaf arrays. In Figure 20, when the amount of keys is smaller than 8M, the space consumption of ROART is larger than P-ART because most of the entries in leaf arrays are empty and much space is wasted. As the amount of keys increases and empty entries are filled, ROART consumes less space than P-ART. Under extreme cases, if each leaf array only has one valid entry, the space waste of ROART will become serious. We think this case is rare because it is hard to construct. To solve this problem, we can use the approach of ART to provide leaf arrays with various sizes.

7. Recovery and Instant Restart. In Figure 21(a), we test normal recovery time with different key numbers. With 128M keys (about 25 GB in total of tree size), data structure recovery takes 19 seconds. With reclamation of free memory (recovery for DCMM), it takes 27 seconds. In this case, free memory chunks in 195 pages (25 GB in total) are reclaimed.

All metadata of data structure and allocator will be restored after restart. So we introduce *selective metadata persistence* (§3.3.1) and *instant restart* (§3.4.2) to hide the recovery overhead of data structure and allocator respectively. The effects of the two techniques are illustrated in Figure 21(b). The test uses 32M keys and 1/36/72 background reclamation threads respectively. The simulation injects a crash at the 11th second, halting for 5 seconds. After restart, ROART can process requests immediately. The recovery of data structure can be delayed until nodes are accessed. Background threads perform the reclamation of free memory chunks in parallel with foreground threads. Experiments show that more threads accelerate recovery process, but causing a greater impact on the foreground performance.

4.4 Performance of NVM Allocators

We make a comparison in Figure 22 between DCMM and other open-sourced persistent allocators, e.g., PMDK [51], nvm_malloc [52] and Makalu [54]. The test workload is to continuously allocate 64-byte chunks, write and persist them. The results show the performance of each thread. PMDK is slow but scalable, the scalability of nvm_malloc and Makalu is poor. DCMM has better performance and scalability than

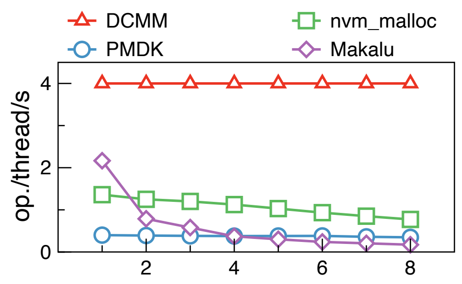


Figure 22: Performance with different allocators.

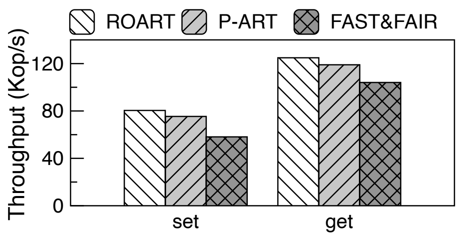


Figure 23: Evaluations in Memcached.

Makalu because DCMM allocates larger pages in a lock-free manner in the first layer, while the page size is only 4K in Makalu and the design in its first layer is lock-based.

There are some other persistent allocators. PAllocator [53] is logging-based with good scalability, but it also suffers extra persistence during allocation/deallocation. PMDK [70] also has a post-crash GC technique, but it is still logging-based. NV-Epochs [22] provides post-crash GC but only supports fixed-sized allocation, and suffers long recovery time. NVM-Reconstruction [55] is a Clang/LLVM extension and runtime library that provides the reconstruction of persistent heaps. Ralloc [56] improves the performance of allocators with post-crash GC, but it still needs a blocked recovery process.

4.5 Real-World System Evaluation

We modify Memcached 1.4.17 to replace its hash index to three persistent indexes, e.g., ROART, P-ART and FAST&FAIR, for persistent storage. We use `memtier_benchmark` to test the performance of set and get operations with single thread. In Figure 23, ROART can outperform P-ART and FAST&FAIR by up to 1.07× and 1.38× in set operations, 1.06× and 1.19× in get operations. The evaluation confirms our previous experiments.

5 Related Works

ROART well resolves the three practical aspects mentioned in §2. Many other related works not mentioned before have also made a lot of efforts.

Persistent Tree-based Indexes. CDDS Tree [1] firstly proposes a multi-version persistent B+Tree design, using copy-on-write techniques without overwriting the original entry, but suffers heavy persistence overhead. DPTree [12] proposes a method to batch modifications in DRAM buffer to reduce persistent overhead, but it needs a background merging process which may stall foreground requests and consume extra bandwidth of NVM. μtree [13] focuses on tail latency in persistent indexes, which is an orthogonal work.

Persistent Hash Indexes. Persistent hash indexes can support fast point access, but has difficulties for range queries. Level hashing [15] proposes a novel two-level hash table structure, reducing the overhead of resizing. Clevel hashing [18] is the multi-thread version of level hashing, based on a lock-free manner and concurrent resize operation in the background. CCEH [16] proposes a three-layer structure based on extendible hashing to reduce NVM writes. Dash [17] uses optimistic concurrent control to improve the parallelism of CCEH, and proposes bucket load balancing strategy to improve load factor of hash table.

Universal Conversion. The general method usually gives a simpler solution to achieve persistent indexes, but less optimization for performance. Izraelevitz et al. [21] design an approach transforming any non-blocking transient data structure to a non-blocking durable one, by adding flush and fence instructions after read or store instructions, but suffering heavy persistence overhead. David et al. [22] propose a `link-and-persist` method to implement log-free concurrent data structures and guarantee durable linearizability. But it can only be applied to 8-byte store/CAS instructions.

6 Conclusion

This paper firstly analyzes three practical aspects, including functionality, performance and correctness. Then ROART is proposed with several optimizations, i.e., (i) *leaf compaction*, (ii) *entry compression*, (iii) *selective metadata persistence*, (iv) *minimally ordered split*, and (v) *instant restart*. Finally, evaluations indicate that ROART can outperform other state-of-the-art indexes by 1.17∼8.27× under various workloads.

Acknowledgments

We are grateful to our shepherd, Vijay Chidambaram, and the anonymous reviewers for their constructive comments and suggestions. This work is supported by National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61877035, 61433008, 61373145, 61572280).

References

[1] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST*, volume 11, pages 61–75, 2011.

[2] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[3] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: reducing consistency cost for NVM-based single level systems. In *13th USENIX Conference on File and Storage Technologies FAST 15)*, pages 167–181, 2015.

[4] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386. ACM, 2016.

[5] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, 2018.

[6] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. BzTree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[7] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. Building Scalable NVM-based B+ tree with HTM. In *Proceedings of the 48th International Conference on Parallel Processing*, page 101. ACM, 2019.

[8] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment*, 13(4):574–587, 2019.

[9] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+ trees: optimizing persistent index performance on 3dx-point memory. *Proceedings of the VLDB Endowment*, 13(7):1078–1090, 2020.

[10] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. clfB-tree: Cacheline Friendly Persistent B-tree for NVRAM. *ACM Transactions on Storage (TOS)*, 14(1):5, 2018.

[11] Ping Chi, Wang-Chien Lee, and Yuan Xie. Making B+-tree efficient in PCM-based main memory. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 69–74. ACM, 2014.

[12] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. Dptree: differential indexing for persistent memory. *Proceedings of the VLDB Endowment*, 13(4):421–434, 2019.

[13] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. *Proceedings of the VLDB Endowment*, 13(11).

[14] Pengfei Zuo and Yu Hua. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST)*, pages 1–10, 2017.

[15] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.

[16] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, 2019.

[17] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, 2020.

[18] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812, 2020.

[19] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, 2017.

[20] Wen Pan, Tao Xie, and Xiaojia Song. Hart: A concurrent hash-assisted radix tree for dram-pm hybrid memory systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 921–931. IEEE, 2019.

[21] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*, pages 313–327. Springer, 2016.

[22] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386, 2018.

[23] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472. IEEE, 2018.

[24] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 462–477. ACM, 2019.

[25] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, 2017.

[26] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, page 4. ACM, 2015.

[27] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *ACM SIGPLAN Notices*, volume 53, pages 28–40. ACM, 2018.

[28] Hyungjun Oh, Bongki Cho, Changdae Kim, Heejin Park, and Jiwon Seo. Anifilter: parallel and failure-atomic cuckoo filter for non-volatile memories. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[29] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 329–343. ACM, 2017.

[30] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 757–773, 2020.

[31] Intel Optane DC Persistent Memory Module. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[32] MySQL. https://www.mysql.com/.

[33] PostgreSQL. https://www.postgresql.org/.

[34] Peloton. https://db.cs.cmu.edu/projects/peloton/.

[35] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.

[36] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[37] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 677–689, 2015.

[38] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1675–1687, 2016.

[39] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment*, 10(2):49–60, 2016.

[40] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35, 2017.

[41] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.

[42] RocksDB. https://rocksdb.org/.

[43] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1077–1091, 2020.

[44] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[45] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment*, 13(7):1091–1104, 2020.

[46] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13, 2011.

[47] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[48] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, 2020.

[49] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, volume 13, pages 38–49, 2013.

[50] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, page 3. ACM, 2016.

[51] PMDK. https://pmem.io/.

[52] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory Allocation for NVRAM. *ADMS@ VLDB*, 15:61–72, 2015.

[53] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment*, 10(11):1166–1177, 2017.

[54] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *ACM SIGPLAN Notices*, volume 51, pages 677–694. ACM, 2016.

[55] Nachshon Cohen, David T Aksun, and James R Larus. Object-oriented recovery for non-volatile memory. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–22, 2018.

[56] Wentao Cai, Haosen Wen, H Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L Scott. Understanding and optimizing persistent memory allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, pages 60–73, 2020.

[57] Zhaoguo Wang, Hao Qian, Haibo Chen, and Jinyang Li. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, pages 1–7, 2013.

[58] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[59] libvmmalloc. https://pmem.io/vmem/libvmmalloc/.

[60] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, pages 473–488. ACM, 2018.

[61] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. HOT: a height optimized Trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 521–534. ACM, 2018.

[62] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196. ACM, 2012.

[63] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[64] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

[65] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 47(4):105–118, 2012.

[66] James L Peterson and Theodore A Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[67] Memcached. https://http://memcached.org/.

[68] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42:34–40, 2017.

[69] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[70] pmemobj. https://pmem.io/pmdk/manpages/linux/master/libpmemobj/pmemobj_first.3.