RWORT: A Read and Write Optimized Radix Tree for Persistent Memory

Jinlei Hu, Zijie Wei, Jianxi Chen*, Dan Feng
Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System of MoE.
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.
{*hujinlei,zjwei,chenjx,dfeng*}@hust.edu.cn

***Abstract*—Tree index structures are widely employed in modern storage systems to support high-performance queries. Persistent memory (PM) brings a new opportunity and challenge for tree indexes. Among persistent tree indexes, we find the radix tree is more suitable than B-Tree for the byte-ability of PM. However, the hierarchy of radix remains excessively high, resulting in high read latency. Node splitting imposes a significant overhead on PM. To address these challenges, we propose RWORT, a read and write optimized radix tree for PM. The key focus of RWORT is to minimize random access in PM and provide efficient write operations. RWORT proposed a hierarchical compression mechanism to significantly reduce the tree height. Additionally, RWORT incorporates mini bloom filters to reduce unnecessary access on PM. For efficient write operations, RWORT uses the lazy split flag and the double-linked pointers to reduce the critical path delay. Furthermore, RWORT introduces a low-overhead ring-based bit tree allocator that improves allocation efficiency on PM. Our experiments show that RWORT improves up to 1.62x/4.91x respectively compared to the state-of-the-art radix tree/B-Tree. RWORT also exhibits higher performance in real-world storage systems such as Memcached.**

I. Introduction

Persistent memory (PM) offers byte addressability, power-down non-volatility, storage capacities comparable to flash media, and performance levels similar to DRAM. Although B-trees are widely used in storage systems, radix trees are gaining popularity due to their advantageous properties. Firstly, radix trees eliminate the need for key comparisons as the tree structure is determined by the prefixes of the inserted keys. This feature makes radix trees well-suited for supporting variable-sized keys. Secondly, radix trees avoid high-overhead and blocking re-balance operations by employing copy-on-write techniques for node insertion or deletion. Similar to individual key-value pairs, only a single 8-byte atomic write operation is required, which improves efficiency, particularly for PM. Thirdly, the height and structure of a radix tree are solely determined by the distribution of key-value pairs. In contrast, B+-trees increase in height as the number of key-value pairs grows, and their structure is influenced by the order of the key-value pairs [6], [8]. However, radix trees based on persistent memory encounter the following challenges:

This work was supported by NSFC No.61821003 and No.U22A2027, the Open Project (HK202201317) of Key Laboratory of Ministry of Industry and Information Technology for Basic Software and Hardware Performance and Reliability Measurement. The Corresponding author is Jianxi Chen.

1) uncompacted tree structure. We have observed a distinct hierarchical pattern in existing radix tree designs [3]–[8]. The root node and its sub-nodes demonstrate full occupancy in general, while intermediate nodes may contain partially contiguous and fully occupied nodes. These un-merged nodes result in an increased height within the radix tree, thereby introducing additional pointer access overhead in the intermediate nodes. Furthermore, the high latency of random reads on PM further diminishes query efficiency.

2) inefficient split operation. Existing studies try to reduce the height of radix trees by employing various node sizes [3]–[5]. However, these approaches still suffer from high-latency node-splitting operations. Radix trees exclusively maintain key-value pairs in their leaf nodes, necessitating traversal to these leaf nodes whenever intermediate nodes split. It results in significant split delays. Additionally, the allocation of PM for splitting incurs additional high overhead.

To solve the two challenges, we present RWORT, a read and write optimized radix tree designed for leveraging the characteristics of persistent memory. The design incorporates three distinct strategies to reduce the tree's height, shorten the search path, and minimize internal node splitting, thereby improving insertion performance. In summary, we have made the following contributions to this paper:

- We propose a match-based strategy to reduce the height of the radix tree, thereby improving its performance. Additionally, we employ a mini bloom filter to reduce access on PM without incurring extra space overhead.
- We utilize a lazy split flag and a double-linked pointer to minimize access on leaf nodes when an intermediate node splits. The double-linked pointers also enhance the efficiency of range scans. Furthermore, we developed a fast memory allocator designed for radix trees.
- We demonstrate that RWORT outperforms the state-of-the-art persistent radix such as P-ART, WORT, and ROART through extensive experiments. Furthermore, it exhibits comparable range-search capabilities to B+Tree.

II. Design

A. Optimal for reading.

Match-Based Strategy To Reduce Height. We argue that the radix tree index exhibits significant hierarchical clustering properties based on evaluations. The higher the node level,

2576-6996/23/$31.00 ©2023 IEEE
DOI 10.1109/ICCD58817.2023.00038

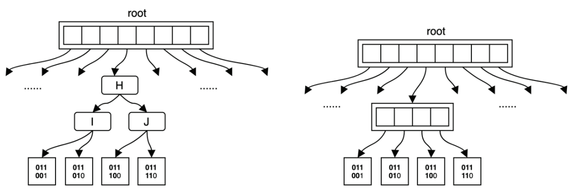


Fig. 1: Internal Node Compression

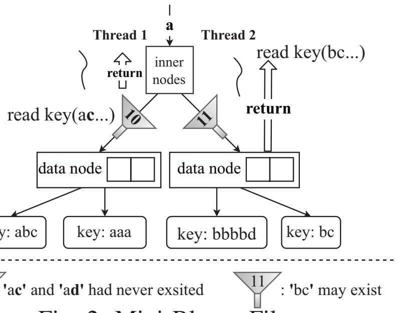


Fig. 2: Mini Bloom Filter

the closer it is to full load. To address these properties, RWORT employs a matching-based strategy to merge continuously distributed fully loaded nodes into a larger node. By compressing nodes at different levels of the radix tree as shown in Figure 1, RWORT effectively reduces the tree's height while keeping the high load factor. RWORT initially compresses the top three saturated layers of nodes into a single and flattened large node. This compression technique shortens the search path without incurring additional space wastage. Additionally, RWORT selectively combines nodes with contiguous distribution during the merging process for internal nodes, thereby reducing the height of sub-trees. This modification not only enhances query performance but also improves cache-friendly.

Mini Bloom Filter To Reduce Accesses. We can not only reduce the tree height by merging nodes but also shorten the number of nodes visited by bloom filter [2]. The bloom filters introduce time and space limitations when applied to tree index nodes. To address these issues, this paper introduces a Mini Bloom Filter (MBF), which is stored in the pointer of the radix tree index's internal node, occupying the highest eight bits. This approach not only minimizes space wastage but also effectively avoids the performance overhead associated with negative queries that access PM. To enhance the multi-thread scalability of the index structure, RWORT adopts a design strategy in which the internal index nodes are stored in DRAM. It also mitigates the tail delay effect associated with storing the entire index structure directly in PM.

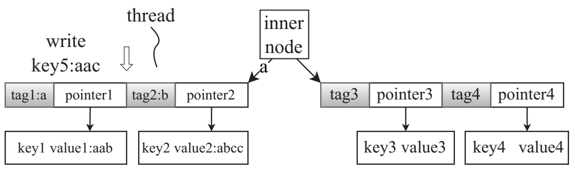
The specific design of the MBF is illustrated in Figure 2. Assuming four types of characters to be inserted: 'a', 'b', 'c', and 'd'. The MBF requires 2 bits of storage. If the first bit is set, it indicates that the key with the next character 'a' or 'b' has been stored in the leaf node. If it is not set, it means that there is no key with the next character 'a' or 'b'. The second bit follows the same pattern indicating 'c' or 'd' charater. For thread 1 in Figure 2, when accessing a key with the prefix 'ac', the search path follows the left subtree in the figure. The decision is made based on the MBF encapsulated in the pointer. If the value is '10', it implies that the string key starting with 'ac' or 'ad' cannot exist, and the search terminates at the MBF. Thread 2 accesses a key starting with 'ba', the search path leads to the right subtree in the figure. In this case, the value of the MBF is '11', indicating that all bits are set and the corresponding charactier may exists.

B. *Optimal for write.*

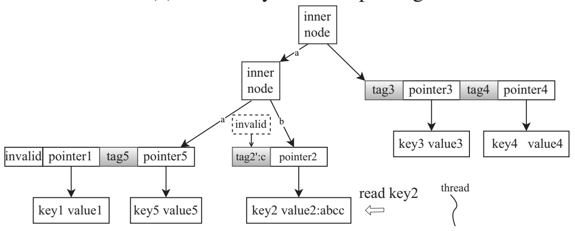
Lazy Split Flag To Reduce PM Access. To reduce overhead caused by the node split operation, ROART employs the lazy split flag technology. In ROART, 8-bit split marks are compressed into pointers to minimize space consumption. It is important to note that the pointer address typically contains only 48 bits of effective address, with the remaining 16 bits utilized for MBF and lazy split flag, respectively.

Assume the leaf node can accommodate a maximum of two entries. When inserting key5 into the tree, as shown in Figure 3a, the leaf node's space is fully utilized, necessitating a split operation. The lazy split flag, encapsulated in the pointer, represents the tag that signifies the next character of the corresponding key. For instance, if the first character of value1 is 'a', the lazy split flag will be 'a'. This indicates that there is no need to access key1 during the split operation, and it can be placed in the fanout corresponding to the character 'a' after the split. Once the split is completed, as shown in Figure 3b, the split flags associated with the leaf node become invalid, and key1 and key2 are eliminated. If a subsequent split occurs, the corresponding keys need to be accessed, and they will be placed in the appropriate positions determined by the specific characters after the split. It is important to note that the lazy split flag is not a one-time use, and it can be employed in subsequent splits. In cases where specific data needs to be accessed (e.g., during query operations with consistent fingerprint comparisons), if the lazy split flag fails, it can be reconstructed after reading. Furthermore, since the lazy split flag is encapsulated in an 8-byte pointer, the split flag can be atomically restored and used in subsequent splits. When a thread accesses key2 in Figure 3b, the split tag 'tag2' associated with key2 is restored upon completion. Thus, during the next split of the leaf node containing key2, the lazy split flag can continue to be used, accelerating the split process

Ring Bitmap Tree Designed For RWORT. The Intel PDMK library provides support for persistent memory allocation; however, the performance of this interface for persistent memory allocation is suboptimal [1]. To address the insufficient in PM allocation and mitigate performance degradation caused by multi-thread conflicts, RWORT employs a two-layer persistent memory allocator called the Ring Bitmap Tree Allocator (RBT). This design avoids the low space utilization associated with linked list organizations. RBT features a two-level structure comprising a global allocator and per-thread local allocators. The global allocator allocates space by utilizing Compare-and-Swap (CAS) to update the latest allocated address in the fixed metadata area of persistent memory and ensures persistence. It allocates a large block



(a) Write key5 cause splitting



(b) Invalid flag after splitting can be rebuilt

Fig. 3: Lazy split flag

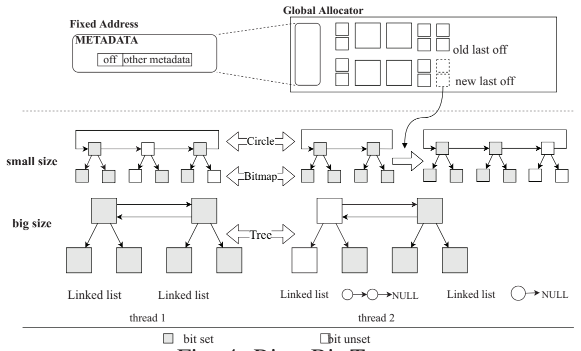


Fig. 4: Ring Bit Tree

of contiguous memory to minimize lock overhead caused by frequent allocations. The local allocator in each thread is a tree-organized bitmap, arranged linearly in address to minimize cache line accesses. Several small trees are managed based on different persistent memory block sizes.The number of trees is typically set to 10 in our experimental environment, which aligns with the number of threads at the maximum bandwidth of concurrent writes to PM. When requests of different block sizes arrive, memory is allocated in the smallest tree that can satisfy the size requirement, usually aligned to the nearest power of 2. The corresponding bit position in this subtree is then set to 1. If all bits in the tree are set to 1, a block of memory is obtained from the global allocator and initialized for subsequent memory allocations, corresponding to the size of the current application block for that thread. RWORT uses a linked list for reclaiming persistent memory addresses not presented in the CBT.

Double Link Pointer For Efficient Range Scans. RWORT sets the bidirectional link before and after the leaf node, and each node contains 64 key-value pairs of data. On the one hand, the next pointer can avoid the pointer chasing problem caused by in-order traversal during the range query process. On the other hand, After adding the backward pointer, you can directly access the previous pointer, reducing the time for inserting data to split. The space overhead of two pointers is negligible compared to the benefit.

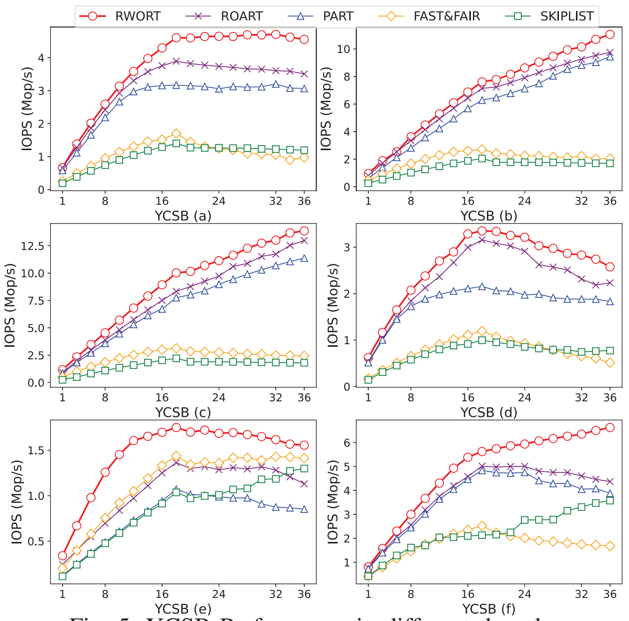


Fig. 5: YCSB Performance in different threads.

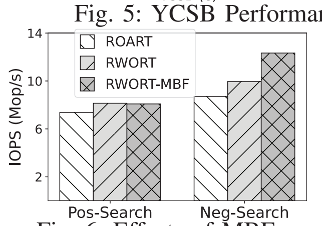


Fig. 6: Effects of MBF.

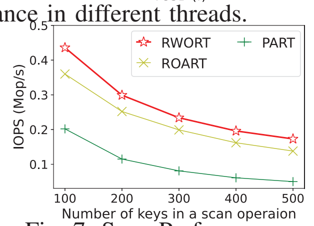


Fig. 7: Scan Performance.

III. EVALUATION

Our experiments are performed on a Ubuntu server 20.04, equipped with two Intel Xeon Gold 6240 CPUs. The system has 4×32 GB of DDR4 DRAM and 4×128GB of *Optane DCPMM* configured in the App Direct mode. In our evaluation, threads are pinned to NUMA node 0. All the code is compiled using GCC 9.5 with all optimization enabled. We comprehensively compared four recent representative index structures: FAST&FAIR, SKIPLIST, PART, and ROART. Among these, FAST&FAIR represents a state-of-the-art B+ tree index based on PM, SKIPLIST is a lock-free skip list structure that incurs no log overhead on PM, PART is a persistent ART-ROWEX based on the principle RECIPE, and ROART is a state-of-the-art radix tree optimized for range queries on PM. The test comprises YCSB workloads, where the key size ranges evenly from 8B to 128B. YCSB-a/b/c/d/e/f represent R50-R50/W5-R95/R100/W100/W5-S95/R50-U50 respectively. The prefix characters W, R, S, and U before the numbers stand for the operations *insert, read, scan, and update after read*, respectively. The key does not include control characters. The value size is set to 8B, a commonly used size in DBMS, consistent with the pointer size. To reflect real persistent memory performance, a single thread is initially employed to insert 20M data into the index structure in the load phase, filling the L3 cache. All tests run for 20 seconds in the run phase, and the average results are calculated.

YCSB-Workloads. For load (a), RWORT demonstrates a

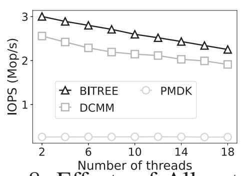


Fig. 8: Effects of Allocator.

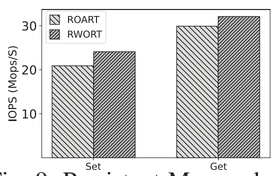


Fig. 9: Persistent Memcached

throughput rate 1.30x higher than ROART under 36 threads. It also exhibits a significant improvement ranging from 1.48x to 3.81x compared to the other three indexes. This performance gain can be attributed to the adoption of compression strategies, which effectively reduce tree height and minimize data writes. Load (b) tests reveal that RWORT achieves 1.13x to 6.51x the performance of the other four indexes with 36 threads. Since this load focuses on read-intensive operations, the outcomes align with those of load (c). Specifically, under read-only test load (c), RWORT outperforms the other four indexes by 1.20x to 5.91x in 36 threads. In write-only load (d), all index structures reach their peak performance at 18 threads. Beyond 18 threads, their performance declines, with the other index structures experiencing a decrease of 23% to 41% due to the NUMA effect. Nonetheless, RWORT maintains a superiority of 15% to 400% over the other four indexes in 36 threads. Load (e) continues to demonstrate the superior performance of RWORT, with its throughput being 1.22x to 6.75x higher than the other four indexes. This is attributed to RWORT's utilization of faster internal nodes from ART stored in DRAM, which further accelerates its operations and employs B-tree leaf nodes for efficient range queries. Lastly, in load (f), RWORT maintains its leading position, outperforming the other four index structures by 12% to 954% in the 18-thread test, demonstrating its scalability advantages under the hybrid architecture.

Mini Bloom Filter. If the internal node of ART is not found, the search process terminates but the B-Trees and skip-list do not. Hence, we will not compare their behavior in this test. RWORT performs similarly to ROART in positive search due to the time-consuming leaf nodes as shown in Figure 6. Before and after turning on MBF, RWORT increased by 1.14x/1.42x compared to ROART. The former improvement is due to the hybrid architecture and compression algorithm, which enhance the efficiency of internal node traversal. The MBF in DRAM effectively reduces the number of negative search requests directed toward persistent memory.

PM Allocators. The test conducted a latency comparison of space allocation using three different persistent memory allocators. RBT and DCMM demonstrate increased performance by 15x/9x times compared to the PMDK library in Figure 8 . Additionally, RBT offers additional optimizations by reducing the number of allocations, utilizing bitmaps to minimize space usage, and aligning cache lines wherever possible. These improvements contribute to enhanced allocator performance when compared to DCMM.

Range Scans. In comparison to ROART, RWORT exhibits an improvement of 18% to 25% in Figure 7, particularly when the number of range query keys reaches 500. Because it reduces the access cost of nodes traversed in order in ROART. RWORT also benefits from its compression strategy, resulting in a lower tree height. Additionally, the double-linked pointers enable direct access to the next leaf node during range queries, eliminating the need for in-order traversal and reducing the number of cache line accesses.

RWORT in Memcached. We evaluate the application of RWORT in Memcached, a real caching system to show the efficient index performance of RWORT. In the single-thread test scenario, applying RWORT in memcached results in a 1.16x/1.09 improvement in set/get operations, compared to ROART as shown in Figure 9.

IV. Conclusion

We propose RWORT, an optimized radix tree for PM, minimizing random access in PM and providing efficient write operations. RWORT significantly reduces the tree height by introducing a hierarchical compression mechanism. Additionally, RWORT incorporates mini bloom filters and persistence optimization to further enhance read operations. Furthermore, RWORT introduces the ring-based bit tree allocator that improves allocation efficiency on PM. Finally, RWORT reduces the critical path delay through the use of a lazy split flag and ensures fast recovery and split operations with the help of double-link pointers. Extensive evaluations demonstrate the superiority of RWORT compared to state-of-the-art radix trees and B-Tree variants.

References

[1] "pmem/pmdk: Persistent Memory Development Kit." https://github.com/pmem/pmdk (accessed Dec. 23, 2022).

[2] "Space/time trade-offs in hash coding with allowable errors — Communications of the ACM." https://dl.acm.org/doi/10.1145/362686.362692 (accessed Jun. 18, 2023).

[3] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013), in ICDE '13. USA: IEEE Computer Society, Apr. 2013, pp. 38–49. doi: 10.1109/ICDE.2013.6544812.

[4] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The ART of practical synchronization," in Proceedings of the 12th International Workshop on Data Management on New Hardware, in DaMoN '16. New York, NY, USA: Association for Computing Machinery, Jun. 2016, pp. 1–8. doi: 10.1145/2933349.2933352.

[5] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "HOT: A Height Optimized Trie Index for Main-Memory Database Systems," in Proceedings of the 2018 International Conference on Management of Data, in SIGMOD '18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 521–534. doi: 10.1145/3183713.3196896.

[6] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems," presented at the 15th USENIX Conference on File and Storage Technologies (FAST 17), 2017, pp. 257–270.

[7] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: converting concurrent DRAM indexes to persistent-memory indexes," in Proceedings of the 27th ACM Symposium on Operating Systems Principles, in SOSP '19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 462–477. doi: 10.1145/3341301.3359635.

[8] S. Ma et al., "ROART: Range-query Optimized Persistent ART," presented at the 19th USENIX Conference on File and Storage Technologies (FAST 21), 2021, pp. 1–16.

[9] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B+-tree," in Proceedings of the 16th USENIX Conference on File and Storage Technologies, in FAST'18. USA: USENIX Association, Feb. 2018, pp. 187–200.