2023 IEEE 41st International Conference on Computer Design (ICCD) | 979-8-3503-4291-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCD58817.2023.00080

Heart: a Scalable, High-performance ART for Persistent Memory

Liangxu Nie[1], Shengan Zheng[2*], Bowen Zhang[1], Jinyan Xu[1], Linpeng Huang[1*]
[1]Shanghai Jiao Tong University, Shanghai, China
[2]MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University, Shanghai, China
[*]Corresponding author: Linpeng Huang, Shengan Zheng
Email: {nieliangxu, shengan, bowenzhang, xujinyan2022, lphuang}@sjtu.edu.cn

***Abstract*—Concurrent indexes for persistent memory (PM) have been extensively investigated due to the appealing features of PM, such as data persistence and DRAM-comparable performance. Among them, Adaptive Radix Tree (ART) is a widely used index that performs well on variable-sized keys. Nevertheless, existing persistent ARTs still suffer from high PM overhead as well as inefficient concurrency control.**

In this paper, we present Heart, a persistent ART with low latency and high scalability. Heart proposes a unified node structure for different capacities with *Hashed Node* and *Node Decoupling* to reduce the PM access overhead. To achieve high scalability, especially in write-intensive scenarios, we propose an efficient concurrency control protocol with lock-free basic operations and lock-free node split. Furthermore, Heart employs *Perceivable Transformation* to avoid anomalies during node expansion and shrinkage. Compared to other state-of-the-art persistent ARTs, Heart achieves up to $21.6\times$ higher performance under YCSB workloads with high memory utilization. We also deploy Heart in DRAM and obtain up to $33.4\times$ speedup than the original in-memory ART.

I. Introduction

Persistent memory [2] has gained wide attention due to its persistence guarantee and DRAM-comparable performance. The latest generation of Optane DCPMM further strengthens cache persistence by introducing enhanced asynchronous DRAM refresh (eADR) [4] technique, which opens up new possibilities for the design and application of PM systems.

In recent years, numerous persistent indexes [1], [3], [11] have been proposed. In particular, the Adaptive Radix Tree [7], [8], which is advantageous in the variable-length KV storage by efficiently organizing strings with similar prefixes, has also been redesigned to better adapt to PM [5], [6], [9]. However, these persistent ARTs share two notable drawbacks.

First, existing persistent ARTs have significant PM access overhead. They mainly focused on reducing the costly cache-line flush instructions, which were used to ensure data persistence in previous PM systems, and are no longer necessary with the emerging cache persistence technique. Moreover, they neglected the excessive PM accesses caused by ART's structural design. As an in-memory index, ART is initially designed with a series of complicated nodes to accelerate basic operations and improve space utilization [7]. However, these nodes tend to store metadata, child node keys and child node pointers in separate areas. Such spatial dispersion of node contents can lead to a larger memory access area during node access. When ARTs are used in PM, the memory access overhead issue leads to significant performance degradation due to the relatively lower performance of PM compared with DRAM. For example, in ROART [9], searching or inserting an 8-byte child pointer in an internal node may result in the reading or writing of up to 3 XPLines (256-byte, the access granularity of physical media in Optane PM [10]), leading to higher latency and increased bandwidth consumption.

Second, ART suffers from inefficient concurrency control, which has been ignored in existing ART-based research. ROWEX [8], the mainstream synchronization protocol of ART, uses node-level write locks, leading to poor scalability in write-intensive scenarios. Furthermore, since reads are not locked, persistent ARTs that use ROWEX as a synchronization mechanism typically use *non-temporal store* instructions (write directly to PM, bypassing the cache hierarchy) to prevent unpersisted data from being globally visible, but this approach fails to leverage the benefits of CPU caching. Another concurrency control scheme, *Optimistic Lock Coupling* (OLC) [8], can cause massive restarts when there are intensive conflicts, further degrading performance. To fully realize ART's potential, it is crucial to improve its concurrency control protocol to achieve higher scalability.

To address the above two issues, we propose Heart, a persistent ART with low latency and high scalability. To lower PM overhead, we propose a unified and PM-friendly structure for internal nodes of different capacities with *Hashed Node* structure and *Node Decoupling* mechanism, reducing operation latency and saving PM bandwidth. *Hashed Node* evenly divides internal nodes into multiple consecutive logical buckets of XPLine size. By hashing the current key byte, a single node operation is contained within the same bucket, so that the PM access of this process does not exceed one XPLine, regardless of the node size. We further propose *Node Decoupling* to decouple the metadata from internal nodes. It enables most updates to be completed with a single write to an 8-byte field called node `context`, which reduces the metadata access overhead and facilitates Heart's efficient

This work is supported by National Key Research and Development Program of China (Grant No. 2022YFB4500303), National Natural Science Foundation of China (NSFC) (Grant No. 62227809), the Fundamental Research Funds for the Central Universities, Shanghai Municipal Science and Technology Major Project (Grant No. 2021SHZDZX0102), and Natural Science Foundation of Shanghai (Grant No. 22ZR1435400).

2576-6996/23/$31.00 ©2023 IEEE
DOI 10.1109/ICCD58817.2023.00080

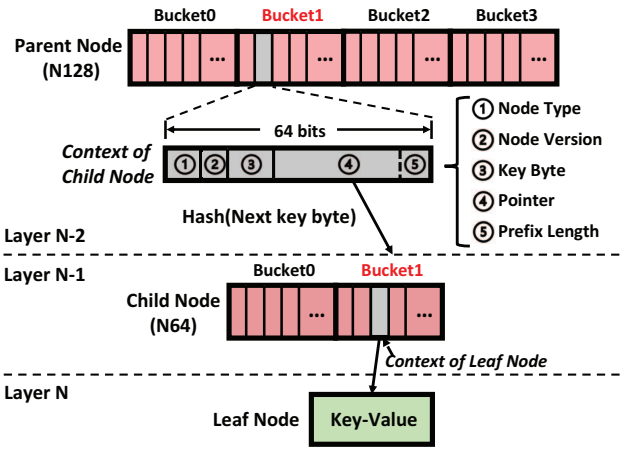

Fig. 1: The node structures in Heart.

synchronization scheme.

To achieve high scalability, especially in write-intensive scenarios, we propose efficient concurrency control methods for Heart. We employ a lock-free approach for all basic operations and node splits by serializing node updates using Compare-And-Swap (CAS) on the 8-byte `context`. Meanwhile, we take advantage of the cache persistence on the latest PM platform to prevent concurrent threads from reading unpersisted data during lock-free access [11]. To address the concurrency anomalies such as *lost update* and *inconsistent read* caused by node transformation, we propose *Perceivable Transformation*, enabling concurrent operations to perceive the occurrence of node transformation and take proactive measures to prevent anomalies.

We implement Heart and our evaluation results show that Heart outperforms other state-of-the-art persistent ARTs and B$^+$-Tree by up to 21.6× under YCSB workloads with high memory utilization. Furthermore, due to the similarity between DRAM and PM indexing brought by eADR, Heart can also be deployed in DRAM and improve the performance in write-intensive scenarios by up to 33.4× in our experiments.

II. Heart Design

A. Unified and PM-friendly Structure

In general, Heart is a persistent variant of the original ART [7] that inherits ART's path compression, node splitting and prefix-based indexing approach. In the following, we first propose a unified node structure with *Hashed Node*. It enables internal nodes, regardless of size, to trigger only a petite range of PM access at a time. Then we propose *Node Decoupling*, decoupling metadata from nodes, which avoids extra PM access for metadata and enables most of the writes to be completed by updating an 8-byte node `context`. Thus, during access of each layer, PM read is contained within one XPLine and PM write is limited to one 8-byte atomic update.

Hashed Node. The core idea of *Hashed Node* is to restrict the access area of a node basic operation (insert, update, delete or search a child pointer) to a small bucket through hash mapping in nodes of different sizes, thus reducing PM access overhead while traversing internal nodes. As shown in Figure 1, an internal node is initially partitioned into several contiguous logical buckets. The bucket size can be set differently depending on the physical media. For example, each bucket is one XPLine (256-byte) size when applied in Optane PM, or one cacheline (64-byte) size when applied in DRAM. Figure 1 also illustrates how *Hashed Node* works between a pair of parent-child nodes. Before accessing the child node content from its parent node, the operation thread first hashes the current key byte and determines which bucket it belongs to. All subsequent lookups and updates will always occur in the selected bucket regardless of the node size, without access requests exceeding one XPLine in PM.

Node Decoupling. Even with hashing mapping of the child data in internal nodes, reading and writing the metadata can still result in additional XPLine accesses. Therefore, *Node Decoupling* is further proposed to optimize the way node content is read and updated by decoupling all the metadata of a node into its parent node in the upper layer. As shown in Figure 1, we encapsulate the necessary metadata together with the address information into an 8-byte field, called the `context` of an internal node. The `context` includes: ① node type (3 bits), ② node version (5 bits), used for concurrency control, ③ key byte (8 bits), the 1-byte key prefix for child indexing, ④ child pointer (42 bits, which is sufficient since all nodes are 64-byte aligned), the address of child node, ⑤ prefix length (6 bits), the compressed path length of the child node. Then, we fill the `contexts` of internal nodes into their parent nodes. In other words, every internal node in Heart is completely composed of its children's `contexts` without any other content or metadata. When inserting a child node into an internal node, we first construct the `context` using the information of the child node and then fill it into an empty slot of the internal node. When looking up a child node pointer in an internal node, we can obtain both the address and metadata of the target child node simultaneously through its `context`, and then enter the corresponding bucket of the child node to continue the search process in the next layer. In this way, access to the next layer is completely restricted to the child node `contexts` without additional reading to the node metadata.

Node Decoupling combines multiple writes during a node update process into an 8-byte update. All the encapsulated metadata is always updated collaboratively with the `pointer` in the node `context` when the node is transformed/split or the slot is redirected. Multiple data updates triggered by these processes can be completed instantly by performing an atomic 8-byte write to the `context`. This also eliminates the possibility of inconsistent states during the write request processing, facilitating our lock-free basic operations/split of internal nodes. This will be further elaborated in Section II-B.

B. Concurrency Control

We propose an efficient synchronization protocol with lock-free basic operations and *Perceivable Transformation*, achieving high scalability especially in write-intensive scenarios.

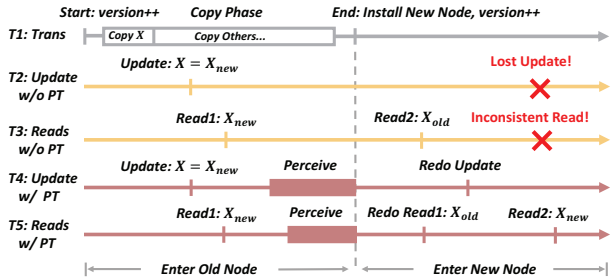

Fig. 2: The anomalies and solutions during node transformation. (PT: *Perceivable Transformation*, X: A child *context* in the node)

Lock-free Basic Operations. In Heart, all basic operations (insert, update, delete or search a child pointer in a target internal node) are performed lock-free. In general, we serialize concurrent writes by updating the target node `context` via the atomic primitive `CAS`. Take an insert as an example, after preparing the new leaf node, and constructing the corresponding leaf node `context`, the worker thread traverses the `context` list in the target bucket with SIMD instructions to find an empty slot and then uses `CAS` to insert the constructed `context` of the new leaf into the empty slot. The insert is committed once the CAS operation succeeds. Otherwise, if the CAS operation fails, the worker thread tries to find another empty slot and repeats the above process. Updates and deletes are executed in a similar manner. Meanwhile, lookups within internal nodes can be executed by traversing the `context` list without blocking or retries since write threads will not make any changes to the target nodes until the updates are atomically committed (i.e., CAS succeeds). The atomic updates in cache-persistent systems guarantee that each visible `context` is consistent and durable at any given moment.

Lock-free Split. Node splitting process in Heart can be executed lock-free. Node splitting is triggered when the inserted key does not match the compressed prefix of the current internal node. It will successively add a new parent node to the current node and then modify its prefix correspondingly. Since we decouple the metadata (including prefix information) from nodes themselves to their parent nodes through *Node Decoupling*, all intermediate results of a split are invisible to other threads until the split is completely finished by atomically updating the target node `context`.

Perceivable Transformation. Heart employs *Perceivable Transformation* to deal with the potential anomalies including *lost update* and *inconsistent read* during node expansion/shrinkage (transformation).

The core idea behind *Perceivable Transformation* is that Heart enables lock-free reads and writes to perceive the occurrence of concurrent node transformations through `version` mechanism and take proactive measures to prevent anomalies. Heart utilizes the node `context` as the medium for transformation threads to share state information with other threads. Specifically, the transformation thread employs `CAS` to atomically increase the node `version` (described in Section II-A) in the corresponding `context` by 1 both before and

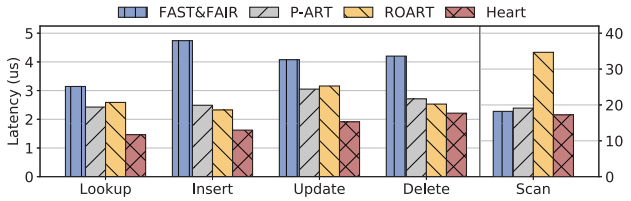

Fig. 3: The latency of base operations under uniform access distribution. (Single thread)

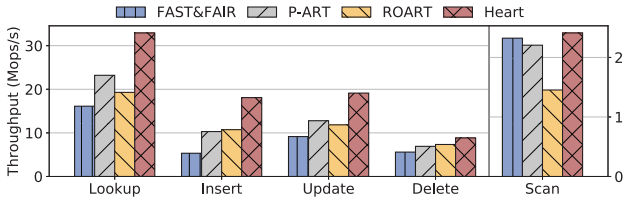

Fig. 4: The throughput of base operations under uniform access distribution. (56 threads)

after executing the transformation. In this way, other threads can determine whether a transformation has occurred or is happening according to the change and parity of the node `version`. When a worker thread plans to access an internal node and obtains its address by accessing the `context` stored in the parent node, it records the current `version` in the meantime. Then it enters the target internal node to perform its operation as described in *Lock-free Basic Operations*. Once the thread has finished its operation on the internal node, it returns to the `context` to read the latest `version`. If a concurrent transformation is perceived (the `version` has changed), the operation thread will wait for the ongoing transformation to complete. Afterwards, the worker thread re-reads the `context`, enters the new internal node and restarts the entire process. T4 and T5 in Figure 2 illustrate how Heart deals with the concurrency anomalies with the proposed *Perceivable Transformation*.

III. Evaluation

A. Experiment Setup

All evaluations use a dual-socket Dell R750 server with two Intel Xeon Gold 6348 processors (28 cores) supporting eADR and 4TB Barlow Pass (BPS) DIMM. We choose two state-of-the-art concurrent persistent ARTs (P-ART [6] and ROART [9]) and one state-of-the-art persistent B$^+$-Tree (FAST&FAIR [3]) to compare the performance with Heart. We warm up each persistent ART with 100 million key-value pairs. Then each test runs for 20 seconds for different workloads and reports the average results. By default, random/skewed keys are generated with sizes between 4 to 32 bytes. Values are fixed at 8 bytes, which can represent indirect pointers.

B. Micro-benchmarks

We begin with the micro-benchmarks to evaluate the latency and throughput performance as well as the space utilization of Heart. We test individual operations using a random key access distribution under single thread and 56 threads before determining the average latency and throughput.

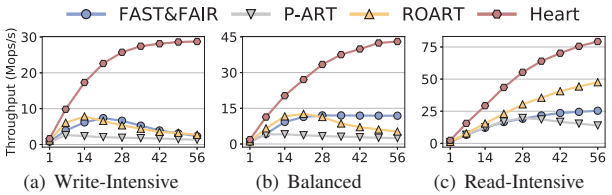

Fig. 5: YCSB throughput with different numbers of threads. (Zipfian access distribution)

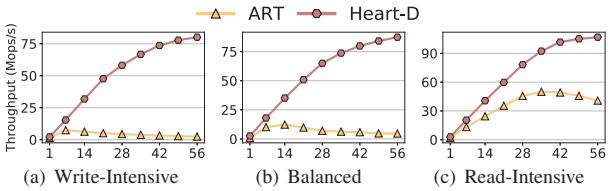

Fig. 6: YCSB throughput in DRAM with different numbers of threads. (Zipfian access distribution)

As shown in Figure 3 and Figure 4, Heart achieves the lowest latency and highest throughput in every base operation. The main reason is that the proposed *Hashed Node* lowers XPLine reads and accelerates traversals, which are the essential steps in all operations. Heart only produces one XPLine read in each layer of internal nodes regardless of node sizes. For write operations, another improvement comes from *Node Decoupling*, which combines multiple PM writes distributed across different XPLines into one 8-byte atomic update. For scan operations, Heart's lead is not significant because the main overhead of scanning is on dereferencing the pointers of leaf nodes rather than the traversal process. In ROART, although *Leaf Array* reduces pointer chasing in internal nodes, it aggregates the leaf nodes with fewer common prefixes and thus increases the number of leaf nodes to dereference, leading to poor scan performance.

For space utilization, after the initialization with 800M variable-sized key-value pairs, Heart consumes 9.561GB of memory space while P-ART consumes 12.924GB and ROART consumes 14.326GB. Heart's lead in space utilization is attributed to the finer-grained series of node sizes (N8, N32, N64, N128, N256) in Heart than the original ART (N4, N16, N48, N256). In addition, the randomness of the hash functions roughly keeps the load balance among different node buckets, which also benefits the space utilization of Heart.

C. Macro-benchmarks

We further evaluate the performance of persistent ARTs with YCSB benchmarks. We generate three workloads in Zipfian distribution with default 0.99 skewness, including (a) write-intensive (10%lookup-90%update), (b) balanced (50%lookup-50%update), (c) read-intensive (90%lookup-10%update).

The results of the four workloads are shown in Figure 5. We observe that Heart achieves near-linear scalability with the increase of threads until exhausting the PM write bandwidth, while other persistent ARTs and FAST&FAIR suffer severe performance degradation. In the write-intensive scenario, other persistent ARTs only scale up to 14 threads due to the node-level write locks. We attribute the high performance of Heart on skewed workload to its efficient concurrency control mechanism, which elides the use of locks for both reading and writing with the proposed lock-free basic operations, lock-free node split and *Perceivable Transformation*. Moreover, the low PM access brought by Heart's PM-friendly structure also saves PM bandwidth and enhances its scalability.

Due to the similarity between DRAM and PM indexing brought by eADR, Heart can also be deployed in DRAM. We implement Heart-D, the DRAM version of Heart with the bucket size set to the *cacheline* size. We compare Heart-D against the original ART [7] with ROWEX [8] under the YCSB workloads. The results are shown in Figure 6. We observe that the lock-free base operations in Heart-D scale well under high contentions. In contrast, the original in-memory ART still suffers from its inherent inefficient concurrency control. Its write performance drops dramatically with the increase of threads because of the node-level write conflicts under skewed workloads. Due to DRAM's lower latency and higher bandwidth compared to PM, Heart-D's lead in DRAM is even greater than Heart's lead in PM, achieving $33.4\times$ higher throughput in write-intensive scenarios.

IV. Conclusion

This paper presents a high-performance persistent ART called Heart, to overcome the high PM overhead and low scalability of existing persistent ARTs. Heart employs a unified and PM-friendly node structure with an efficient concurrency control mechanism. Compared to other state-of-the-art persistent ARTs, Heart achieves significantly higher performance in both PM and DRAM.

References

[1] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. In *VLDB*, 2020.

[2] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 2017.

[3] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *FAST*, 2018.

[4] Intel. eadr: New opportunities for persistent memory applications. https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html, 2021.

[5] 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 *FAST*, 2017.

[6] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *SOSP*, 2019.

[7] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, 2013.

[8] 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*, 2016.

[9] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. Roart: Range-query optimized persistent art. In *FAST*, 2021.

[10] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *FAST*, 2020.

[11] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. Nbtree: a lock-free pm-friendly persistent b+-tree for eadr-enabled pm systems. In *VLDB*, 2022.