CANRT: A Client-Active NVM-Based Radix Tree for Fast Remote Access

Yaoyao Ying[1], Kaixin Huang[1], Shengan Zheng[2], Yaofeng Tu[3], and Linpeng Huang[1(✉)]

[1] Shanghai Jiaotong University, Shanghai, China
{yingyy77,Kaixinhuang,Lphuang}@sjtu.edu.cn
[2] Tsinghua University, Beijing, China
venero@tsinghua.edu.cn
[3] ZTE Corporation, Nanjing, China
tu.yaofeng@zte.com.cn

Abstract. This paper presents the first study of building a remote-accessible persistent radix tree, named CANRT. Unlike prior works that only focus on designing single-node tree structure for non-volatile memory, we focus on optimizing remote access performance for a persistent radix tree while minimizing the persistence overhead. Simply adopting server-reply paradigm will incur heavy server CPU consumption and hence lead to high operation latency under concurrent workloads. Therefore, we design a low-latency *node-oriented read* mechanism and a *fine-grained lock-based write* mechanism to minimize the server CPU involvement in the critical path. We also devise a *non-blocking resizing* scheme in CANRT. The extensive experimental results on commercial Intel Optane DC Persistent Memory platform show that CANRT outperforms the state-of-art server-centric persistent radix trees by 1.19x–1.22x and 1.67x–1.72x in read and write latency, respectively. CANRT also gains improvement of 7.44x–11.15x in terms of concurrent throughput under YCSB workloads.

Keywords: Radix tree · Non-volatile memory · RDMA · Data consistency · Concurrent access

1 Introduction

Emerging storage and networking technologies such as non-volatile memory(NVM) and Remote Direct Memory Access (RDMA) technology are poised to reshape conventional data storage and memory systems in data centers, bringing new opportunities to achieve efficient remote data storage and access. Non-volatile memory technologies such as PCM [17], STT-RAM [16] and the recently released Intel Optane DC Persistent Memory [8] promise non-volatility, byte-addressability and DRAM-comparable latency. RDMA technology makes it possible for clients to access remote memory region while bypassing server CPU and kernel, which contributes to high bandwidth and low latency.

M. Qiu (Ed.): ICA3PP 2020, LNCS 12452, pp. 433–447, 2020.
https://doi.org/10.1007/978-3-030-60245-1_30

To further exploit the advantages of NVM and RDMA technology, indexing data structures and algorithms are required to be carefully redesigned to promise both data consistency and data concurrency for higher scalability. In recent years, a large number of tree-based indexing structures are proposed for non-volatile memory. Most of these works focus on the design of persistent B+-tree, [1,7,14,19]. They achieve considerable performance by reducing the number of expensive memory fencing and cache line flush operations.

Lee et al. [12] propose three radix tree variants (i.e., WORT, WOART, ART+CoW), and demonstrate that radix tree is more appropriate for persistent memory than B+-tree variants due to its unique features. For instance, the radix tree does not demand tree rebalancing operations and each insertion or deletion only results in a single atomic update operation to the tree, which is perfect for NVM [12]. However, whether the B+-tree variants or the radix tree variants in existing papers are designed for NVM in a single machine environment instead of distributed systems. Therefore, when deployed in distributed system environment such as data centers, they may suffer from high access latency and poor throughput scalability. Even though they can be equipped with RDMA server-reply paradigm [2,5,9,13], where both reads and writes are processed by the server and requires replies to the clients, the limited server CPU resource will become the bottleneck for highly-concurrent remote requests and impair the overall performance. The server-bypass merits of RDMA cannot be exploited with such designs.

In this paper, we propose a client-active NVM-based radix tree for fast remote access with server-bypass paradigm using RDMA. In the server side, we decouple the radix tree into two parts: data nodes that store actual key-value pairs with the same prefix, and prefix nodes that locate the target data nodes. Each data node contains an 8-byte metadata, which is used to resolve concurrent write collisions. Specifically, prefix nodes are also stored in all clients to support server-bypass remote data access. Our contributions can be summarized as follows.

- We propose CANRT, a client-active and NVM-optimized radix tree that leverages RDMA technology to speed up remote data access. To the best of our knowledge, CANRT is the first persistent radix tree that is optimized for fast remote access.
- We decouple the radix tree nodes into prefix nodes and data nodes to enable server-bypass remote data access and utilize a bitmap-assisted strategy for all types of writes to efficiently guarantee data consistency while resolving concurrent write collisions.
- We design a node-oriented remote read mechanism and a fine-grained lock-based remote write mechanism to support low-latency and high-concurrency remote access to the persistent radix tree. We also propose a non-blocking resizing scheme for CANRT to boost its performance during tree reconstruction.
- We conduct extensive experiments for CANRT. The results show that CANRT outperforms the server-centric persistent radix trees by 1.19x-1.22x for remote reads, 1.67x–1.72x for remote writes and 3.10x–3.83x for range

query in terms of latency. For concurrent throughput, CANRT outperforms the counterparts by up to 11.15x under write-intensive workloads.

Organization. The rest of this paper is organized as follows. Section 2 introduces the background and motivation of our work. Section 3 describes the design and implementation of CANRT. Section 4 presents experimental results and we conclude this paper in Sect. 5.

2 Background and Motivation

2.1 Non-Volatile Memory

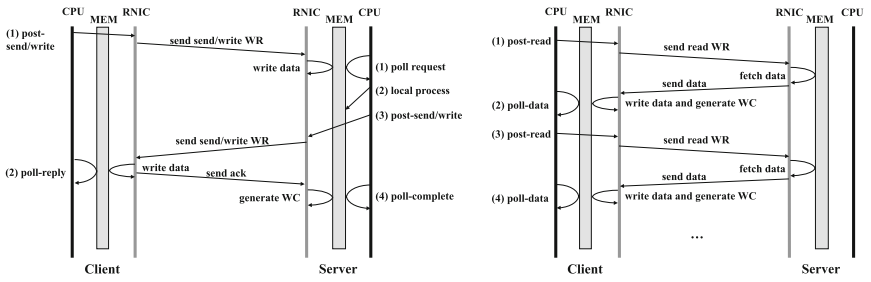
Emerging memory technologies such as phase change memory(PCM) [17], spin-transfer torque RAM (STT-RAM) [16] and the recently released Intel Optane DC Persistent Memory [8], can be directly accessed through the memory bus via processor loads and stores, delivering DRAM-like performance while providing persistent data storage [3,15]. NVM provides potential to design efficient persistent index structures [1,7,12,14,19], storage systems [2,3,11,15,18] and etc, with leveraging its non-volatility, byte-addressability and DRAM-comparable latency. Data consistency is a significant issue for NVM as there may be partial writes and reordering writes during system crash, which will lead to inconsistent data state and unexpected system error. The mainstream solutions are using logging or copy-on-write techniques to guarantee data consistency [3,11,15]. The persistence ordering is maintained by using $clflush/clwb/clflushopt$ and $mfence$ instructions.

2.2 Remote Direct Memory Access

Remote direct memory access (RDMA) is a new networking technology that supports low-latency, high-bandwidth, zero-copy and kernel-bypass access to remote memory region. Since Reliable Connection (RC) provides full support for RDMA semantics, in this paper, we mainly focus on the RC mode of using RDMA.

Normally, there are two types of RDMA semantics: message semantics and memory semantics. Message semantics employ SEND/RECV verbs for user-level message exchange. Due to the requirement of the matching RECV request posted at the server side, these verbs are generally referred as two-sided RDMA operations. Memory semantics, on the other hand, are one-sided semantics. These semantics utilize READ/WRITE verbs and are proved to achieve higher throughput and lower latency than message semantics [10]. Adopting these semantics is attractive in that they free server from sending requested data to clients, replying acknowledge information or coordinate collisions, as well as continuously polling for the incoming requests [5,6,9,13].

Apart from above semantics, RDMA also support one-sided atomic operations, including FETCH_AND_ADD (FAA) and COMPARE_AND_SWAP (CAS). Both of these verbs operate on data with 64 bits and are atomic operations relative to other operations on the same NIC. This paper utilizes FAA to support concurrent remote write operations to the radix tree.



(a) Server-Reply Read/Write (b) Server-Bypass Read

Fig. 1. Remote access mechanisms for current tree-based indexing structures

2.3 RDMA-enabled Index Structure

A few recent researches have used RDMA technology in key-value stores to accelerate the remote data access [2,5,6,9,13]. However, in these designs, the fundamental indexing structures are all hash tables. In fact, few researches have been proposed to optimize the current tree-based indexing structures using RDMA to fit in distributed environment for efficient remote data access. This can be attributed to three challenges.

First, *unawareness of data location*. Since traversing through an indexing tree means the retrieval of data at multiple memory addresses, it is relatively harder to locate the target data than hash tables. Second, *data consistency*. The employment of NVM raises consistency issues. While tree structure is often more complicated compared to hash table, it demands more careful design to avoid such problems. Third, *concurrent write collision*. Since the write operations to a tree tend to affect multiple nodes, concurrent writes are easy to conflict, which as a result can harm the concurrent performance.

2.4 Persistent Radix Tree

Most of the previous proposed persistent trees are variants of B+-tree, such as NVTree [19], wB+tree [1], FPTree [14] and FAST&FAIR [7]. Lee et al. [12] proposed three variants of a radix tree: WORT, WOART and ART+CoW. It is pointed out that radix tree is actually more appropriate for NVM compared to B+-tree variants. For instance, key comparisons in a radix tree are no longer required since the radix tree structure is decided only by each character of the inserted keys. Furthermore, updates in node granularity and expensive tree rebalancing operations are also avoided. The experimental results of [12] show that the radix tree variants outperform most of the current B+-tree variants(i.e., NVTree, wB+Tree and FPTree).

Although WORT has already improved the indexing performance on NVM, it cannot be directly deployed in an RDMA-enabled distributed system environment since it is designed for a single machine environment. Fortunately, similar

to previous hashtable-based key-value stores [5,9,13], we can apply the server-reply paradigm as a general RPC mechanism for WORT to support remote read and write requests. However, there are two performance limitations: 1) high server CPU overhead/high latency. Since data access through a radix tree means the traversing of the tree, clients require the server to coordinate and process requests to access target data. Taking read request as an example, there are two methods to execute it which are illustrated in Fig. 1. One method is to let the server collect all the requested keys and send back to the client (shown in Fig. 1(a)). Although it can be performed in 1 RTT, the server CPU overhead will be quite high for concurrent access and thus incurs high operation latency. The other is to allow the server to only reply addresses to the client and it is the client's responsibility to complete the reads using one-sided READ verb (shown in Fig. 1(b)). It is almost impossible for clients to directly access data in the server side within 2 RTTs, which also leads to high operation latency and heavy bandwidth consumption. As a result, either significant server CPU overhead or high operation latency will be incurred. 2) poor concurrency. Concurrent access is demanded in most distributed database systems or key-value store systems. Unfortunately, WORT does not support concurrent access requests, which is harmful to both system throughput and latency performance.

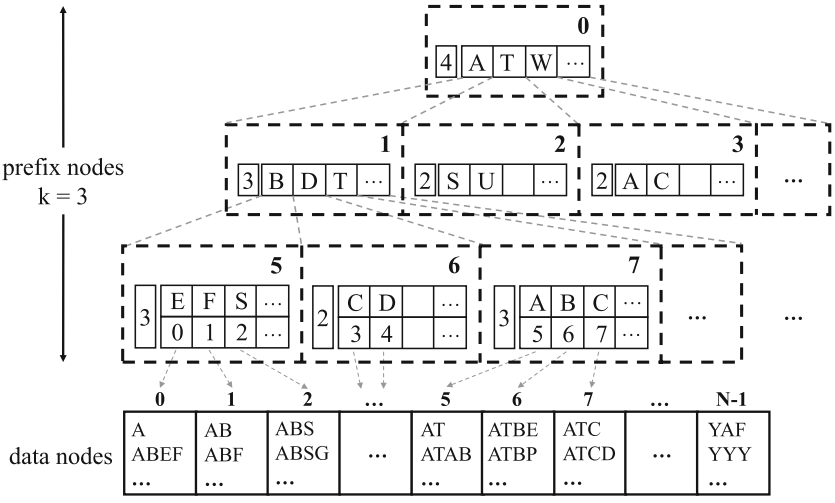


Fig. 2. CANRT storage architecture

3 CANRT

3.1 CANRT Data Structures

Figure 2 depicts the server-side storage architecture of CANRT. Basically, CANRT is composed of two major parts:

Prefix nodes, which exist in both the server and clients, are organized in a pointer-less format. In CANRT, all the prefix nodes are pre-allocated in a consecutive memory space in the server, and will be sent to each client when the tree is reorganized. Figure 3 (a) and Fig. 3 (b) illustrate the layouts of prefix nodes. CANRT does not store explicit keys in its prefix nodes. Instead, a prefix node consists of an array of characters, each of which represents one character of the inserted keys. Additionally, each prefix node contain an *nChars* field, which records the number of valid characters stored in that node.

Data nodes, which exist only in the server and contain actual key-value pairs, can be accessed through the node ID stored in the last level of the prefix nodes. Each key-value pair in the data node is encapsulated in a DataItem entry and the DataItems in each data node are kept unsorted. As depicted in Fig. 3(c) and Fig. 3(d), to both accelerate data access and enable high write concurrency, an 8-byte metadata structure is adopted in the data node. The metadata contains a bitmap and a write lock.

In CANRT, only the first k characters of the inserted keys will be stored in the prefix nodes, which we call the prefix of the key. Here, k also denotes the height of the prefix nodes. Figure 2 shows an example of a CANRT with $k = 3$. Taking the key ATCD as an example, the prefix is ATC, and each character of the prefix (i.e., A, T, C) is stored in each level of the prefix nodes. Particularly, in the last level of the prefix nodes, apart from the character array, there is another array (denoted as NodeID array) storing the corresponding data node ID (i.e., C is stored in the third level of the prefix nodes, and a data node ID 7 is stored in the corresponding position of the NodeID array which means that keys with the prefix of ATC are stored in the eighth data node).

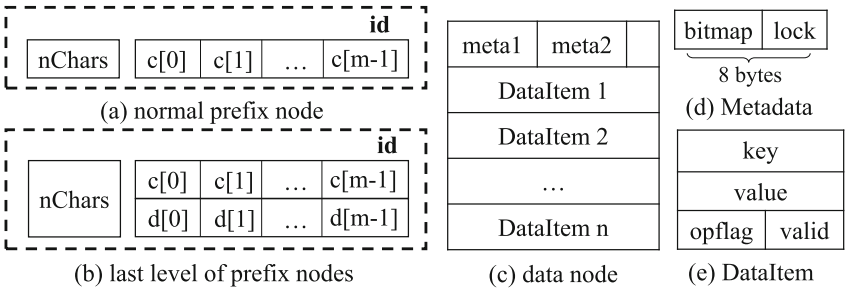


Fig. 3. CANRT node layout

3.2 Locating Target Data Node

Both read and write operations in CANRT need to find the target data node first. In CANRT, the position of each prefix node is fixed and the node ID of

each prefix node increases sequentially from zero. Hence it can be used to locate the target prefix node in each level and finally find the target data node ID.

Generally, the i_{th} $(i < k)$ character of the search key will be used to probe the target node in $(i+1)_{th}$ level, and the k_{th} character will finally locate the data node which contains the search key. For instance, with each prefix node contains m characters, the i_{th} level of prefix nodes contains m^{i-1} nodes and the base node ID in $(i+1)_{th}$ level (denoted as b_{i+1}) can be calculated by

$$b_{i+1} = b_i + m^{i-1} \tag{1}$$

where b_i is the base node ID in i_{th} level. Based on the recursive relations in (1), we can calculate the value of b_i by the following equation:

$$b_i = \frac{m^{i-1} - 1}{m - 1} \tag{2}$$

Now we can locate the target node ID in $(i+1)_{th}$ level (denoted as I_{target}) using the equation below:

$$I_{target} = b_{i+1} + (I_{cur} - b_i) * m + p \tag{3}$$

where I_{cur} denotes the node ID of the current node in i_{th} level and p is the position of the i_{th} character of the search key in the character array of the current node. When $i = k$, the current node is in the last level of prefix nodes, and the data node ID can be directly obtained from the NodeID array of current node with the position p.

In the cases when the i_{th} character of the request key does not exist in the corresponding character array, the search process will be terminated and the target data node ID will be set as the node ID of the last data node. Once the target data node ID is obtained, the address of the target data node can be calculated based on the memory address of the first data node and the size of each data node.

3.3 Fine-Grained Lock-Based Remote Write

To support high-concurrency remote writes, we design a fine-grained lock-based remote write mechanism. It employs server-bypass paradigm using one-sided RDMA verbs and makes append-only updates to the server's persistent region in order to avoid any data corruption. There are three phases in remote write procedure, all of which are client-active: locking phase, checking phase and writing phase. The whole remote write procedure is provided in Algorithm 1.

In the locking phase (lines 1 to 9), the client first calculates the target data node ID with locally-buffered prefix nodes. Then it uses a FETCH_AND_ADD verb to obtain the 8-byte metadata of the target data node from the server, while atomically adding 1 to the *lock* field of metadata. If the fetched *lock* value is greater than 0, it means that the target data node has been locked by another client or the server itself. In such cases, the client will wait for random backoff

Algorithm 1. remote_write(key, value)

```
1: /* locking phase: fetch the metadata of target data node */
2: node_id ← findDataNode(key);
3: raddr ← data_base + datanode_size * node_id + shadow_meta_off;
4: len ← 8;
5: post_rdma_fetch_and_add(raddr, len, laddr)
6: meta ← laddr;
7: if meta.lock > 0 then
8:    retry from step 5 after random backoff;
9: end if
10: /* checking phase: find a free bit in the bitmap */
11: for bit in meta.bitmap do
12:    if bit == 0 then
13:       /* writing phase: write a new DataItem to the target node */
14:       generateDataItem(OPFLAG, key, value, 1);
15:       raddr ← raddr+meta_size+bit_index * dataitem_size;
16:       len ← dataitem_size; imm ← (node_id << 8 | bit_index);
17:       post_rdma_write_with_imm(raddr, len, laddr, imm);
18:       return
19:    end if
20: end for
21: imm ← (node_id << 8 | RESIZE_FLAG);
22: post_send_with_imm(imm);
23: retry from step 1 after being notified a resizing accomplishment;
```

time (e.g., 1 us), after which the lock operation will be replayed. Otherwise, current client successfully locks target node and enters the checking phase.

In the checking phase (lines 10 to 12 and 19 to 23), the client will verify if there are free bits (i.e., 0-bit) in the *bitmap* field. If there is one or more free bits, the client proceeds to the writing phase. Otherwise, it indicates that the target node is full (i.e., there is no free DataItem slot), the client will inform the server to resize the prefix nodes (see Sect. 3.5) and resending the request after the server informs the accomplishment of resizing.

In the writing phase (lines 13 to 18), the DataItem location corresponding to the first free bit in the *bitmap* will be regarded as the target location to append a new write. For all types of remote write operations, including insert, delete and update, the write steps in the client side are almost the same, except the $opflag$ value in each packed DataItem to be appended. The $opflag$ indicates the specific operation type. For example, insert, delete and update can be represented by 1, 2 and 3, respectively. The client then immediately transmits the newly-generated DataItem to the target location in the server using a WRITE_WITH_IMM verb. The carried 32-byte immediate informs the server of both the node ID of the target data node and the specific location of the newly inserted DataItem in the data node. From the perspective of clients, a remote write operation is completed once the WRITE_WITH_IMM request has been successfully posted, which incurs no server CPU involvement in the critical path.

Algorithm 2. remote_read(key, find_value)

```
1: /* fetch both metadata and data*/
2: node_id ← findDataNode(key);
3: raddr ← data_base + datanode_size * node_id;
4: len ← datanode_size;
5: post_rdma_read(raddr, len, laddr);
6: datanode ← laddr;
7: (meta, data) ← datanode;
8: /* local search */
9: find_value ← NULL ;
10: for i ← 0, i < dataitem_num, i++ do
11:    if meta.bitmap[i] == 1 && data[i].key == key then
12:       find_value ← data[i].value;
13:       return find_value;
14:    end if
15: end for
16: return NULL;
```

In the server side, a background thread will process the appending messages from clients, which is out of the critical path of client request execution. When polling a work completion (WC), it will first locate the newly inserted DataItem according to the immediate data, and then checks the *opflag* field to determine the write operation type. For insert or delete operation, the server simply atomically modifies the metadata of the data node by changing the corresponding bit (i.e., 0 to 1 for insert and 1 to 0 for delete) in the *bitmap* and resetting the *lock* field to 0. For update operation, the server will mark the old DataItem as invalid and validate the newly inserted DataItem at the same time with two bit flips. For all types of write operations, the server will set the *valid* field in the DataItem to 0 once the metadata update completes. To maintain persistence ordering for data consistency, we utilize *clwb* and $mfence$ primitives accordingly.

Notice that a remote write operation is only visible after the corresponding metadata in the server is modified, which avoids data inconsistency during system crash. Since the modification of the metadata is an 8-byte atomic write operation, the expensive logging or copy-on-write is unnecessary. The fine-grained lock-based remote write mechanism of CANRT has two merits: 1) it enables clients to write data without the involvement of the server CPU, which considerably reduces the server processing overhead in the critical path; 2) it allows DataItems in each data node to be unsorted and no data shift is required, hence reducing extra writes to NVM.

3.4 Node-Oriented Remote Read

Instead of simply accessing one DataItem, CANRT fetches an entire node from the server-side for each single read. Remote Read operations also start with locating the target data node. As introduced in Sect. 3.2, the target node locating can be conducted in the client directly. After obtaining the target data node ID,

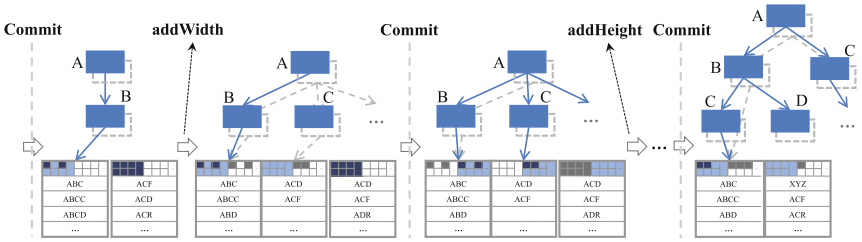


Fig. 4. Non-blocking resizing

the client posts a READ request to fetch the target data node in the server radix tree. Algorithm 2 illustrates the remote read mechanism of CANRT. Since only one RDMA operation is posted, the total network overhead is only 1 RTT. After fetching the required data node to local buffer, the client then checks the *bitmap* entry in the metadata field and scans the valid DataItems to retrieve the matched DataItem.

Although the unsorted DataItems increase the searching time inside the data node, the searching performance is still acceptable since the number of DataItems inside a data node is limited (see Sect. 4). Thanks to the bitmap-assisted strategy, the read operations will never be blocked by any concurrent write operation since the search process will only check the DataItems whose bit value is one in the *bitmap*.

3.5 Non-blocking Resizing

As the size of each data node is fixed, when one data node is full, the server need to allocate more space to hold more keys. As such, existing keys in the tree may be reorganized and migrated to newly-generated data nodes. This procedure is called radix tree resizing. The resizing can be triggered either by the server itself or the notification of a client, as lines 21–22 in Algorithm 1 indicate.

We propose a non-blocking resizing scheme for CANRT. That is, even during data migration, the radix tree in the server can still be accessed by remote clients. The key idea is using a tick-tock design (see Fig. 4) inspired by Redis dictionary [20] to avoid any modifications to current tree structure during data migration. Specifically, CANRT maintains two versions of radix tree, one version as the valid tree and the other as the shadow tree. Any read or write operation is performed on the valid tree and the shadow tree is exclusively designed for resizing.

In CANRT, there are two types of resizing: addWidth and addHeight. When the last data node is full, addWidth resizing is performed to store more characters in each level of prefix nodes. Otherwise, the resizing procedure is launched in the addHeight manner. Both addWidth and addHeight will not affect the old data

nodes, the newly generated data nodes will simply be appended to the end of old data nodes and the DataItem migration will be reflected by the metadata.

With the shadow tree, the resizing scheme in CANRT is non-blocking for read operations, and only a portion of write operations will be blocked due to locking failure to the full data nodes. Compared to read and write operations, the resizing procedure seems time-consuming in large CANRT, but the frequency of resizing can be significantly dropped with more and more prefixes inserted into the prefix nodes (6.20% when inserting one thousand keys and 1.47% when inserting one million keys). We believe that such overhead is acceptable.

3.6 Crash Recovery

Inconsistency may occur when system crashes during remote write operations or resizing procedure and thus recovery is required. The recover procedure starts with detecting the locked data nodes and then checks the *valid* field of each DataItem in those data nodes. The *valid* field is the last bit of each DataItem and indicates both the integrity of the DataItem and whether the server side process for that DataItem is completed. If the *valid* value is 1, which means the DataItem is not processed, the server will simply execute server side metadata update, as discussed in Sect. 3.3. If the valid value is 0, it means current DataItem 1) has already been processed by the server or 2) is an incomplete write from the client. In either case, the server-side data state is consistent.

After fixing the remote write inconsistency, the server will then check the resizing consistency. We use two state flags (i.e., $pxflag$ and $dtflag$) to mark the completion of prefix nodes reconstruction and data migration, respectively. Only when the two flags are both marked, the system crash does not occur during resizing procedure. Otherwise, the recover procedure should be performed. When $pxflag$ is unmarked, it means that the resize procedure just starts, and the server will simply replay the whole resizing procedure. When $pxflag$ is marked and $dtflag$ is unmarked, it means that system crashes after the reconstruction of the prefix nodes but before the completion of data migration. In such cases, the server will perform the data migration recovery. The procedure ends with atomically marking the $dtflag$ which transforms the shadow tree to the new valid tree.

4 Evaluation

4.1 Experimental Setup

Different from previous works that leverage a DRAM-based PM performance emulator (e.g., Intel PMEP/Quartz) to emulate PM latency [1,7,12,14], we conduct our experiments on a small cluster of three machines, which are all equipped with commercial Intel Optane DC Persistent Memory, and Intel Xeon(R) 2.60 GHz CPU (32 KB/1 MB/24 MB L1/L2/L3 cache). Our experiments are performed in the AppDirect mode of AEP, which exposes a separate

(a) Remote Write (b) Remote Read (c) Range Query

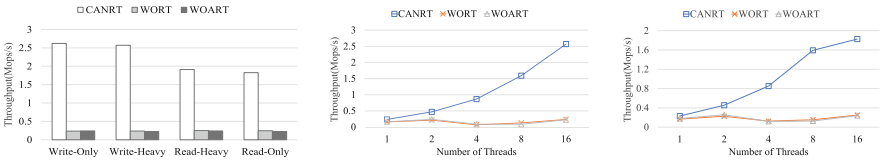
Fig. 5. Remote read and write performance comparisons

persistent memory device. As for RDMA hardware configuration, all of these machines are equipped with Mellanox ConnectX-5 InfiniBand NIC and run on Ubuntu 18.04 (kernel version 4.18.04). For each experiment, we use one machine as server, while the remaining two machines work as clients. Since 16-byte key has been broadly adopted in current key-value stores [18], we set the key and value in CANRT to be 16 bytes and 15 bytes, respectively.

For comparison, we implement both WORT and WOART by adding RDMA mechanism to it. To accelerate remote access, similar to previous RDMA-enabled hash works [5,9], we design a ring buffer mechanism for both WORT and WOART. That is, for each client, the server keeps a request ring buffer and for both remote read and write procedure, clients only need to write requests to the ring buffer using a WRITE_WITH_IMM verb. The server dedicatedly checks the corresponding client ring buffer and makes processing once a request is found. Afterwards, the server will inform the client using a SEND verb. The procedure ends once the client receives the server's reply message. We evaluate the operation latency with random String workload and test throughput with the widely-used YCSB [4] benchmark. Each result is averaged for 10 runs.

4.2 Remote Write Latency

Figure 5(a) shows the comparison of remote write performance among CANRT and the other two radix tree variants under random and sequential workloads. For random workloads, CANRT is roughly 1.7x faster than the other two trees. Although it seems that CANRT requires more RTTs (i.e., about 1.5 RTTs) in one remote write operation, it is steady and little affected by the server's CPU usage. For WORT and WOART, apart from 1 RTT (i.e., 0.5 RTT for client's WRITE and 0.5 RTT for server's SEND), waiting for the server's reply also incurs heavy time consumption, including request polling, key locating, data persistence, and posting reply message. All these processing steps are in the critical path of request execution. Since each operation requires interaction between the client and the server, the latency is more unstable compared with CANRT. Although WOART has better local write performance than WORT, the relatively much higher network overhead makes this difference no longer noticeable. For sequential workloads, the performance of CANRT is dropped by 7.94% compared with random workloads. This is because sequential workloads contribute to a higher probability of data node access conflict.



(a) various workloads (b) write-heavy scalability (c) read-heavy scalability

Fig. 6. Throughput and scalability comparisons under YCSB workloads

4.3 Remote Read Latency

Figure 5(b) compares the remote write latency of three radix trees. We observe that CANRT outperforms its counterparts by 1.19x-1.22x. It is also noticed that the read latency of CANRT is slightly higher than the write latency shown in Fig. 5(a). Although each remote read operation of CANRT only consumes one RTT, the size of the data transferred per operation is larger compared with write operations. For WORT and WOART, the server-reply mechanism incurs higher latency than CANRT due to the processing overhead in the critical path. For sequential workloads, the latency of WORT and WOART is lower than the latency under random workloads. The reduction of server side cache misses under sequential workloads contributes to the improvement of WORT and WOART. Although there is almost no performance improvement for CANRT under sequential workloads, the total latency of CANRT remote read operations is still lower than WORT and WOART.

4.4 Range Query Latency

Since DataItems with same prefix are stored in the same data node in CANRT, we integrate consecutive requests for the same data node into one request, which significantly improves the range query performance. For WORT and WOART, due to the structure limitation, the remote range query functions are simply implemented by calling a remote read request for each key. We collect the range query performance by querying 10000 and 100000 keys respectively, and the results shown in Fig. 5(c) demonstrate that CANRT achieves much better performance for range query. When querying 10000 keys, CANRT is roughly 3.1x faster compared with WORT or WOART. When querying 100000 keys, CANRT is 3.71x faster than WORT and 3.82x faster than WOART.

4.5 Concurrent Throughput

In the experiments shown in Fig. 6(a), we evaluate the concurrent throughput under four types of YCSB workloads: Write-Only (100% update), Write-Heavy (50% read + 50% update), Read-Heavy (95% read + 5% update) and Read-Only (100% read). The experiments are conducted using 16 threads. We can observe that CANRT consistently outperforms the other two radix trees due

to its server-bypass remote access mechanism. For WORT and WOART, since each operation requires the server's process, the server CPU becomes the bottleneck of the remote access. For CANRT, write-only workload performs best among the four YCSB workloads, and read-only workload performs worst. This is because write operations consume less bandwidth. In CANRT, each write operation only transfers one 8-byte metadata and one DataItem while the read operations require fetching the whole data node. However, since each remote read operation only requires one RTT and the number of DataItems in one data node is limited, the remote read throughput is still acceptable.

Figure 6(b) and Fig. 6(c) show the scalability performance of three radix trees under write-heavy and read-heavy workloads, respectively. Due to the server CPU limitation, WORT and WOART exhibit poor scalability and they are unable to saturate the network bandwidth. By contrast, the throughput of CANRT increases with the number of client threads. For Write-Heavy workloads, Fig. 6(b) shows that when the number of threads increases from 2 to 16, the throughput of CANRT is increased by 10.88x, which almost increases proportionally to the number of threads. However, for read-heavy workload, the experimental results in Fig. 6(c) show that the performance of CANRT is increased by a factor of 7.93, when the number of threads increases from 2 to 16, which increases proportionally to the number of threads only when the number of threads is less than 8. The reason is that when the number of client threads reaches 16, the network bandwidth becomes the bottleneck of remote access. Notice that remote read operations carries larger packet each time than remote write and hence the read throughput bottleneck is reached earlier.

5 Conclusion

In this paper, we propose CANRT, a client-active NVM-based radix tree for fast remote access using RDMA technology. We decouple the radix tree nodes into data nodes and prefix nodes to exert the full potential of both NVM and RDMA. We design a node-oriented read mechanism for low-latency remote reads and a fine-grained lock-based write mechanism for fast remote writes. The experimental results show that CANRT performs better than server-centric persistent radix trees in terms of both remote access latency and throughput.

Acknowledgements. This work is supported by the National Key Research and Development Program of China (No. 2018YFB1003302), SJTU-Huawei Innovation Research Lab Funding, and the China Scholarship Council (No. 201906230180).

References

1. Chen, S., Jin, Q.: Persistent b+-trees in non-volatile main memory. Proc. VLDB Endow. **8**(7), 786–797 (2015)
2. Chen, Y., Lu, Y., Yang, F., Wang, Q., Wang, Y., Shu, J.: 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, pp. 1077–1091 (2020)

3. Coburn, J., et al.: Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. ACM SIGARCH Comput. Archit. News **39**(1), 105–118 (2011)
4. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing, pp. 143–154 (2010)
5. Dragojević, A., Narayanan, D., Castro, M., Hodson, O.: Farm: fast remote memory. In: 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), pp. 401–414 (2014)
6. Huang, H., Huang, K., You, L., Huang, L.: Forca: fast and atomic remote direct access to persistent memory, pp. 246–249 (2018). https://doi.org/10.1109/ICCD.2018.00045
7. Hwang, D., Kim, W.H., Won, Y., Nam, B.: Endurable transient inconsistency in byte-addressable persistent b+-tree. In: 16th {USENIX} Conference on File and Storage Technologies ({FAST} 18), pp. 187–200 (2018)
8. Intel: Intel optane dc persistent memory (2019). https://newsroom.intel.com/news-releases/intel-data-centric-launch/
9. Kalia, A., Kaminsky, M., Andersen, D.G.: Using rdma efficiently for key-value services. In: ACM SIGCOMM Computer Communication Review, vol. 44, pp. 295–306. ACM (2014)
10. Kalia, A., Kaminsky, M., Andersen, D.G.: Design guidelines for high performance {RDMA} systems. In: 2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16), pp. 437–450 (2016)
11. Kim, W.H., Kim, J., Baek, W., Nam, B., Won, Y.: NVWAL: Exploiting NVRAM in write-ahead logging. ACM SIGPLAN Not. **51**(4), 385–398 (2016)
12. Lee, S.K., Lim, K.H., Song, H., Nam, B., Noh, S.H.: {WORT}: Write optimal radix tree for persistent memory storage systems. In: 15th {USENIX} Conference on File and Storage Technologies ({FAST} 17), pp. 257–270 (2017)
13. Mitchell, C., Geng, Y., Li, J.: Using one-sided {RDMA} reads to build a fast, CPU-efficient key-value store. In: Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13), pp. 103–114 (2013)
14. Oukid, I., Lasperas, J., Nica, A., Willhalm, T., Lehner, W.: 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, pp. 371–386. ACM (2016)
15. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. ACM SIGARCH Comput. Archit. News **39**(1), 91–104 (2011)
16. Wang, K., Alzate, J., Amiri, P.K.: Low-power non-volatile spintronic memory: STT-RAM and beyond. J. Phys. D Appl. Phys. **46**(7), 074003 (2013)
17. Wong, H.S.P., et al.: Phase change memory. Proc. IEEE **98**(12), 2201–2227 (2010)
18. Xia, F., Jiang, D., Xiong, J., Sun, N.: Hikv: a hybrid index key-value store for dram-nvm memory systems. In: 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17), pp. 349–362 (2017)
19. Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K.L., He, B.: Nv-tree: reducing consistency cost for nvm-based single level systems. In: 13th {USENIX} Conference on File and Storage Technologies ({FAST} 15), pp. 167–181 (2015)
20. Zawodny, J.: Redis: lightweight key/value store that goes the extra mile. Linux Mag. **79**(8), 1–10 (2009)