PFtree: Optimizing Persistent Adaptive Radix Tree for PM Systems on eADR Platform

Rui Zhang, Yao Wu, Shangyi Sun, Lulu Chen, Yibo Huang, Ming Yan, and Jie Wu(✉)

School of Computer Science, Fudan University, Shanghai, China
{zhangrui21,shangyisun21}@m.fudan.edu.cn,
{yao_wu20,llchen18,huangbb16,myan,jwu}@fudan.edu.cn

Abstract. Persistent memory (PM) provides byte-addressability, low latency as well as data persistence. Recently, a new feature called eADR is available on the 3rd generation Intel Xeon Scalable Processors with the 2nd generation Intel Optane PM. eADR ensures that data stored within the CPU caches will be flushed to PM upon the power failure.

In the eADR platform, previous PM-based work suffered more read/write amplification and random access problems, and memory allocations on PM are still expensive. The persistence ways on the eADR platform are still unclear. Therefore, we propose PFtree (PM Line Accesses Friendly Adaptive Radix Tree), a persistent index optimized for the eADR platform. PFtree reduces PM line access with two optimizations: stores key-value pair in leaf array directly to reduce pointer chasing and stores necessary metadata with key-value pair closely and auxiliary metadata in DRAM. PFtree reduces memory allocations in critical paths by allocating bulk memory when creating a leaf array. Then, we design an adaptive persistence way based on data block size for PFtree to fully use PM bandwidth. Experimental results show that our proposed PFtree outperforms the radix tree by up to 1.2× and B+-Trees by 1.1−7× throughput, respectively, with multi-threads.

Keywords: Persistent Memory · Adaptive Radix Tree · eADR

1 Introduction

Emerging byte-addressable persistent memory (PM), such as the Intel Optane DC Persistent Memory Module (DCPMM) [8], is now commercially available. PM is attractive because it offers DRAM-comparable performance and disk-like endurance. In the first generation, PM-equipped platforms support the asynchronous DRAM refresh (ADR) feature [9]. The write content in the CPU cache is still unstable. Therefore, we need to use explicit cache line flush instructions and memory barriers to ensure the persistence of PM writes.

With the arrival of 3rd generation Intel Xeon Scalable processors and 2nd generation Intel Optane DCPMM, extended ADR (eADR) becomes available

X. Wang et al. (Eds.): DASFAA 2023, LNCS 13943, pp. 46–61, 2023.
https://doi.org/10.1007/978-3-031-30637-2_4

[10]. Compared to ADR, eADR further ensures that data in the CPU cache is flushed back to the PM after a crash. It ensures the persistence of globally visible data in the CPU cache and eliminates the need to issue expensive synchronous flushes. The advent of eADR not only omits the flush instruction but also allows us to revisit the design of PM-based data structures.

Building efficient index structures in PM promises high performance and data durability for in-memory databases [4,15,21]. Most existing persistent indexes [1,3–5,12] are designed only for ADR-based PM systems. They work on achieving crash consistency, and the optimization for performance generally lies in reducing the number of flushes, because data flushing is expensive in ADR-based PM systems. However, in the second generation PM, these optimizations are no longer evident as the flush instruction is no longer needed, therefore, these optimizations are not evident when the previous work is applied directly to the second generation PM.

ART(Adaptive Radix Tree) was widely used in database systems because it supported range query and variable-sized keys. Although there have been some works of ART on NVM [11,13,18], there have been some problems. First, the previous work had more PM read/write amplification problems. The previous PM-based data structure usually stored the metadata in the header, separately from the data. Therefore, reading and writing the metadata in the header and the data will cause more PM write amplification problems. Second, the throughput of PM is still slower than DRAM, especially in random access. Separating metadata from data leads to more random accesses [16,17] because the metadata needs to be accessed before the data, and leading to more PM line accesses. Third, the persistence ways on the eADR platform are still unclear. There is no need for persistence instructions on eADR, so there are various ways to achieve data persistence. It is not clear what scenarios these persistence ways are applicable to.

This paper presents PFtree, a PM Line Accesses Friendly Adaptive Radix Tree, to deliver high scalability and low PM overhead. PFtree proposed a leaf array to compress leaf nodes in ART(Adaptive Radix Tree) to solve the problems above. To our knowledge, PFtree is the first PM-based ART optimized for eADR-enabled PM systems.

To reduce PM line accesses, PFtree stores the payload and metadata of each key-value data closely in the leaf array so that when reading or writing a key-value data, it can be done in the same XPline and reduce the PM read/write amplification. In addition, PFtree does not store a pointer to key-value data like others [16,18]. Instead, key-values are stored directly in the data area of the leaf array. For variable key and value, PFtree uses metadata to help store and recovery from the leaf array. The directly stored key-values not only reduce one pointer chasing but also reduce PM line accesses. Meanwhile, PFtree stores auxiliary metadata for accelerated lookup and insertion into DRAM, further reducing PM read and write accesses. These auxiliary metadata are violate and can be reconstructed from other data.

To reduce the overhead of PM memory allocation, PFtree uses bulk memory allocation. The leaf array in PFtree allocates a large chunk of memory at creation time. It then stores the key-value data directly into the memory of the leaf array at insertion time, avoiding the overhead of allocating new memory every time the key value is inserted.

To choose the appropriate persistence way, the performance of the new generation CPU and the new generation PM has been tested in detail. In the new scenario, there are three persistence methods for data persistence to PM. We analyze the latency and bandwidth of the three persistence methods in detail under different data sizes and different threads. Finally, a suitable persistence method was selected for PFtree in different scenarios.

The main contributions are as follows.

- We propose PFtree, the first persistent range index based on eADR platform, to take full advantage of eADR. PFtree not only focuses on data crash consistency but also focuses on reducing PM line accesses to reduce read and write amplification.
- We provide an in-depth analysis of the persistent ways in eADR platforms. Then we propose an adaptive persistent way to fully utilize the PM bandwidth.
- We perform experiments to compare PFtree with state-of-the-art tree-based indexes, including ROART [18], P-ART [13], FAST&FAIR [7], and BzTree [1]. PFtree outperforms the existing solutions by 1.1−7× throughput under YCSB workloads.

2 Background

2.1 PM and eADR

Compared to DRAM, The most significant difference of PM is the non-volatile property. Data written to PM will persist and still exist after power failure. Figure 1 shows the architecture of PM systems and the internal architecture of Optane PM. We assume that the system consists of one or more NUMA-enabled multicore cpus, each with local registers, storage buffers, and caches and that the last level of cache (LLC) is shared between all cores of the CPU. Each CPU has its own memory (DRAM and PM), which is connected to other CPUs through mesh interconnect. PM and Write Pending Queues (WPQ) connect to mesh connect through iMC (integrated memory controller).

Extended ADR (eADR), which is supported in the 3rd generation Intel™ Xeon™ Scalable Processors, solves data consistency problems by making sure that CPU caches are also included in the so called “power fail protected Domain” [10]. In an eADR environment, the CPU cache is also a part of the persistent domain, so there is no need to flush data from the cache to the ADR domain. The data in the CPU cache will be automatically persisted to the PM after a power failure or software crashes. However, the data in the CPU register is still volatile. Because of the nature of eADR, the operations and designs that ensure crash consistency in applications do not need to be performed.

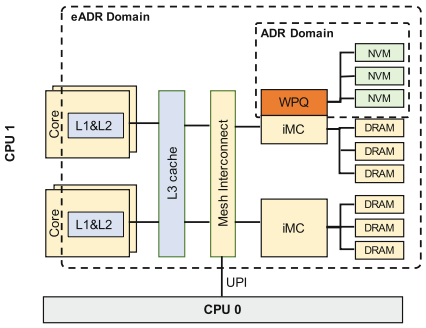

Fig. 1. Architecture of PM.

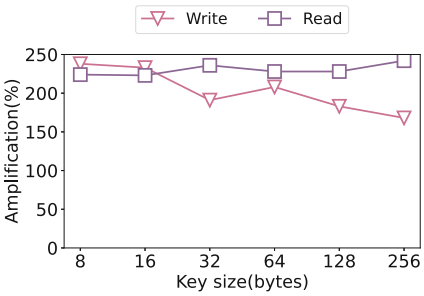

Fig. 2. Read and write amplification in ROART.

2.2 ART and Its Persistent Variants

Adaptive Radix Tree (ART) is a space-efficient radix tree that can dynamically change the node size according to node utilization. To reduce the tree height, ART sets the number of child Pointers in the node to 2^8 and uses a one-byte partial key for each node. At the same time, ART reduces memory consumption by using one of four different node types. Figure 3 illustrates the node structures of ART. Node4 and Node16 store the keys in order together with the corresponding pointers. Sequential search is enough for Node4 because of its small size. SIMD instructions can be used to accelerate the search in Node16. Node48 has a child index with 256 slots to quickly locate the corresponding pointer. Node256 is the normal node in a radix tree whose radix is 256. Starting with Node4, ART adaptively converts nodes to larger or smaller types when the number of entries exceeds or falls behind the capacity of the node type. This requires additional metadata and more memory manipulation than a traditional radix tree, but still shows better performance than other cache conscious in-memory index constructs.

P-ART is a persistent version of the concurrent ART that uses instructions in RECIPE [13] for transformation. For Crash Consistency, P-ART re-used a helper mechanism to detect and fix crash inconsistencies during restarts. However, p-ART does not guarantee dural linearizability-that is, it is possible to read volatile data even if the corresponding operation has returned. ROART [18] is an improved version of P-ART for supporting efficient range queries, lower memory allocation overhead and correctness. However, because ROART inherits the rebalancing algorithm and index structure from ART [14], it still incurs a high allocation overhead during SMOs (structural modification operations), as it needs to allocate more than two leaf arrays for each split operation.

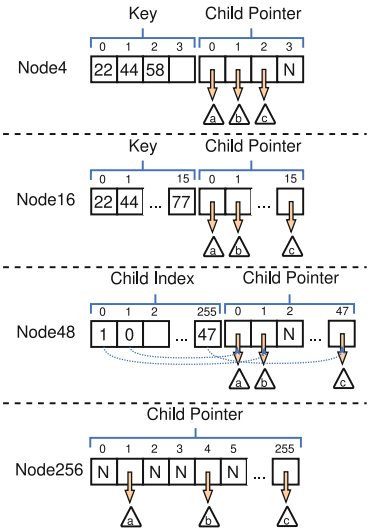

Fig. 3. Node structures in ART

2.3 Motivation

Reduce PM Reading and Writing. In the ADR platform, since the CPU cache is a volatile domain, persistence instructions are needed to ensure data persistence. Much previous work [1,2,7,17,19,20,22] has been devoted to achieving crash consistency and recoverability of data while neglecting optimization of PM line accesses. By analyzing the PM-based indexes, as shown in Fig. 2, ROART has significant read and write amplification to PM under different workloads. One reason is that the metadata in these structures are stored in the head and the key-value data are stored in the tail, and known research work shows that there is 256B buffer inside the PM, so this operation across multiple buffers causes significant read and write amplification.

Reduce PM Memory Allocations on Critical Paths. To improve the efficiency of range queries in ART, ROART [18] uses a leaf array to store pointers to leaf nodes in a compressed manner, in order to eliminate the need to traverse different levels of the trees and pointer chasing. ROART stores each complete key and value in the leaf node, therefore, a memory allocation on PM is made when a key-value pair is inserted. However, the memory allocation in PM incurs high overhead, increasing the query's latency and decreasing the throughput.

Leverage the Features of eADR. The new eADR has multiple methods for persisting data. We are motivated to reveal the effects of these methods on different data sizes, to make full advantage of the eADR features in the real environment.

3 Design

PFtree aims to further optimize query and insertion of ART on PM. Specifically,

- (1) PFtree stores key-value pairs directly in leaf arrays instead of pointers to key-value pairs. It can not only reduce pointer chasing but also reduce PM line accesses.
- (2) PFtree stores metadata and key-value data close together, PM line accesses can be reduced.
- (3) By taking advantage of the latest eADR hardware platform, PFtree's persistence strategy cleverly supports an adaptive persistence way that automatically selects the best persistence strategy according to the block size of the persistent data.

3.1 Block-Based Leaf Array

Motivated by ROART [18], PFtree uses leaf array to delay the leaf split and reduces pointer chasing to improve the range queries in ART. Unlike ROART's leaf array, PFtree stores key-value pairs directly in the leaf array instead of pointers to key-value pairs to further reduce pointer chasing and PM line accesses. In addition to leaf nodes, PFtree does not change the types of other nodes in ROART, that is, the internal nodes will still use Node4, Node16, Node48 and Node256, as mentioned before.

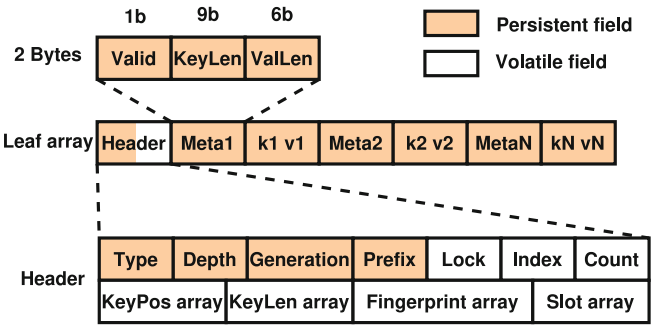

Fig. 4. The leaf array structure of PFtree

Overview of Leaf Array. Figure 4 shows the basic structure of the leaf array in PFtree. The header of each leaf array holds the metadata necessary to identify the characteristics of the leaf array, which can be used for quick recovery and concurrency control. The following is a large data area for storing key-value pairs. In addition to the raw data of key-value pair, there are two bytes of metadata before them, which are used to reconstruct and update key-value from the data area. In the two-byte metadata, the first bit represents the validity of key-value

pair, 0 means deleted and 1 means normal. The next nine bits represent the length of the key. The last six bits record the value length of this key-value pair. After the metadata, we store the raw key-value pair. Although we can traverse and lookup in leaf arrays using only two bytes of metadata, this is slow and inefficient. It is tedious and inefficient to parse the metadata and calculate the position of the next key-value pair each time, so we store the $KeyPosarray$ and $KeyLenarray$ in the header to speed up the traversal and update. Except for volatile metadata, all other data of the leaf array is stored in PM, and the data of the internal nodes are also stored in PM.

The Header of Leaf Array. The header stores the necessary metadata and some other data to speed up queries and updates. *Type* indicates that the type of the node is Node4, Node16, Node48, Node256, or leaf array. *Depth* indicates the depth of the node in the entire PFtree, which facilitates traversal of the entire ART. *Generation* records the generation of each node, for quick recovery after startup. *Lock* is used to mark whether the leaf array is locked, which is used for concurrent access control. $Index$ records the index of the inserted key-value pairs in the subsequent 4 arrays. $Prefix$ records the common prefixes in the leaf array. $Count$ records how many key-value pairs have been inserted into the current leaf array. $KeyPosarray$ and $KeyLenarray$ record the offset of each key-value pair in the data area and the length of the key, respectively. Similarly, $Fingerprintarray$ records the fingerprint of each key-value pair. None of these three arrays needs to be persistent because we can rebuild and update key-value pairs by using the metadata of the first two bytes of each key-value pair in the data area. In addition, we also use *slotarray* to record the size relationship between leaf nodes, which can further improve the efficiency of range queries. Among these metadata of header, $Type$, $Depth$, and $Prefix$ are initialized when the leaf array is created, and *Generation* is persisted and restored after each restart. We do not need to modify them after creation. The remaining violate data can be recovered by scanning the data area. The detailed recovery process can be found in the section Data Structure Recovery.

The Advantages of Leaf Array. *PFtree can reduce pointer chasing and PM line accesses through storing key-value pairs directly.* Key-value pairs are stored directly in PM, we do not need to get the pointer and then get the value according to the address as before. Directly stored key-value pairs can reduce a PM-based pointer chasing and also reduce random access on PM because the data and pointer to it are stored in different places.

Adjacent Storage of Metadata and Key-Value Pair Can Reduce PM Line Accesses. The previous data structure tended to keep the metadata in the header and the payload of key-value pair in the back part of leaf nodes. This can make the metadata storage more compact, but in each lookup or modification, we need to update the metadata in the header and then turn to the tail to manipulate the key-value pairs. When there are many key-value pairs, more PM lines will

be accessed during persistence. In the leaf array of PFtree, as shown in Fig. 4, we store the necessary metadata in the first two bytes of each key-value pair, followed immediately by the key-value pair. The advantage of such adjacent storage is that we do not need to read or write data across multiple PM lines, and the number of reads and modifications in the same PM line can reduce the PM read/write amplification. Also in ADR environment, storing metadata and key-value pair close together can reduce flush times.

Volatile Metadata Stored in DRAM can Reduce PM Line Accesses. In the leaf array of PFtree, metadata such as *KeyPosarray*, *KeyLenarray*, can be recovered by the 2-byte metadata in front of each key-value pair, so no persistence is required. Considering that the read/write latency of PM is still higher than that of DRAM, we store this recoverable metadata in DRAM, thus reducing the read/write access to read PM.

Bulk Memory in Leaf Array Can Reduce the PM Memory Allocation on Critical Path. The leaf array has already allocated a large chunk of memory when it is created, so there is no need to allocate memory again when inserting key-value pair into the leaf array, and only need to copy key-value pair to the free area, thus reducing the PM memory allocation on the critical path.

3.2 PM-Aware Flush Mode

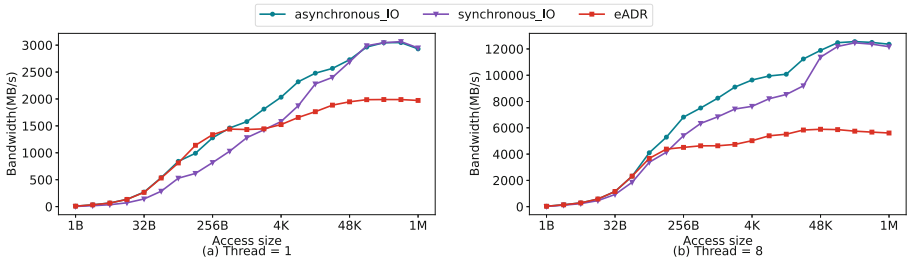

Fig. 5. Random write bandwidths varying with access size for the three persistence ways

With the application of new processors and the new generation of PM, there are three ways to persist data: ❶ `synchronous IO`: Use the persistence instructions the same as before. We can use *clwb* and *fence* to flush data from the L1 cache to PM or use *NTStore* and *fence* to store data directly to PM without going through the CPU cache. ❷ `eADR approach`: Use the features provided by eADR to persist data without using any instructions. In the eADR-enabled system, the CPU cache is located in the persistent domain. As a result, we can persist the data in the L1 cache to PM without instructions like *clwb* and *fence*. ❸ `asynchronous IO`: Use the *flush* to force the data from the cache to PM

without calling the *fence* for synchronization. Traditional PM can not support this way, but we can avoid using the fence while keeping the program correct in the eADR system.

To explore which persistence way can make the most of the PM bandwidth and achieve the lowest persistence latency, we first test the basic performance of PM, explored in different threads, different access sizes, form of persistence time latency and the change of bandwidth. Since most programs are based on random access to PM (such as ART insertion and update operations), we will focus on the latency and bandwidth of random read and write to PM under the three different persistence ways in the following experiments. The hardware configuration of the experiment will be introduced in Sect. 4.

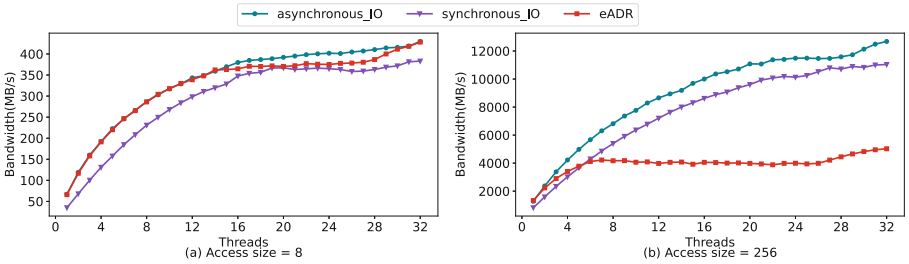

Fig. 6. Random write bandwidths varying with thread for the three persistence ways

Different Access Sizes. We first explore the latency and bandwidth of different persistence ways under different access sizes. We explore the bandwidth of random write with single-thread and multi-thread (8 threads in our experiments) respectively. Figure 5a shows the latency and bandwidth changes of the three persistence ways under different access sizes in the case of a single thread. When the access size is smaller than 128B, the bandwidth of asynchronous IO and eADR persistence is almost the same. In the range of 128B to 256B, the persistent bandwidth in eADR mode is slightly higher than that in asynchronous IO mode. After 256B, the persistent data bandwidth in asynchronous IO mode is higher than that in eADR mode. Especially, when the access size is large, the persistent bandwidth in asynchronous IO mode is significantly higher than that in eADR mode. As for the traditional synchronous IO mode, it has been proven that the fence instruction incurs a significant persistence overhead, so the persistence bandwidth is always lower than the asynchronous IO mode.

In the multi-thread case, the results show similar conclusions to the single-thread case. For example, in the case of 8 threads as shown in Fig. 5b, when the access size is small, the bandwidth of persistent data in asynchronous IO and eADR modes are similar. When the access size exceeds 64B, asynchronous IO bandwidth is higher than eADR bandwidth. As access size continues to grow, the bandwidth gap between the two becomes more and more apparent.

Therefore, it can be seen that when the access size is small, eADR can achieve or even exceed the bandwidth of synchronous and asynchronous IO with less overhead, and as the access size becomes larger, asynchronous IO can get more bandwidth. It still keeps the same pattern in multi-thread. We speculate that the reason for this phenomenon is as follows: when using eADR way for persistence, without the help of any persistence instructions, the persistent data in PM needs to be passively evicted from the cache, from L1 cache to L2 cache to L3 cache, and finally to PM. However, the flush instruction can flush data directly from the L1 cache to PM, which is expected to reduce the IO path of data and reduce the latency of data persistence. For this reason, we propose an asynchronous IO persistence approach that uses flush instructions to omit this part of the data persistence path to reduce data persistence latency.

Different Threads. As for the influence of different persistence modes on PM bandwidth under multi-thread, we also conducted some experiments to analyze. For example, when the access size is 8 bytes, asynchronous IO throughput is similar to eADR throughput when the number of threads is less than 12. Only when it exceeds 14, asynchronous IO performance is slightly better than eADR.

Figure 6b shows the variation of PM bandwidth with different persistence ways when the access size is 256B. When the number of threads is 2, the PM bandwidth in asynchronous IO persistence is higher than that in eADR. As the number of threads increases, The asynchronous IO mode is significantly higher than the bandwidth using eADR mode. This is similar to the previous conclusion: when the access is large, using asynchronous IO persistence can obtain higher PM bandwidth.

Based on Fig. 6, we find that eADR can still obtain high persistence bandwidth without additional persistence instructions when the number of threads and access size both are small. When access size exceeds 256B, asynchronous IO can obtain higher persistence bandwidth. As the number of threads increases, it is more efficient to use asynchronous IO persistence when the access size is large. Based on these findings, we design an adaptive data persistence approach in the eADR environment: the data persistence way is determined by the size of the data.

3.3 Adaptive Flush in Insert and Update

Since eADR platform is gradually on the market, we can take full advantage of eADR features to reduce the overhead of implementing persistent instructions. Based on our previous observations, not all persistence cases that use the eADR approach yield the best performance. Therefore, we design an adaptive persistence approach to take advantage of the low latency of eADR-style while allowing larger chunks of data to achieve higher bandwidth. As shown in Algorithm 1, when persisting data, we first judge the size of the data. If it is a large data block (larger than 256B), the persistence instruction CLWB is invoked to forcibly refresh the data to PM so as to make better use of the bandwidth of

Algorithm 1. flush_data(void *addr, size_t len)

```
1: if support_eADR() then
2:    if len > 256 then
3:       for each cacheline in data do
4:          clwb(cacheline)
5:       end for
6:    else
7:       eadr_auto_flush(data)
8:    end if
9: else
10:   for each cacheline in data do
11:      clwb(cacheline)
12:   end for
13:   fence();
14: end if
```

PM. If it is a small data block (smaller than 256 bytes), eADR mode is used to refresh the data automatically. That is, no persistence command is invoked and the data is passively swapped out of the cache to achieve persistence.

4 Evaluation

Our evaluations consist of four parts to reflect the performance improvements of each proposed design. We evaluate and compare PFtree with some available and usable state-of-the-art representative indexes. For ART, we choose two other state-of-the-art PM ART, ROART [18] and P-ART [13]. We also evaluated the other state-of-the-art B+tree indexes and tire indexes such as BzTree [1] (lock-free B+tree), FAST&FAIR [7](logless crash consistency).

4.1 Experimental Setup

We run experiments on a server with an Intel(R) Xeon(R) Silver 4314 CPU clocked at 2.40 GHz, 512 GB of Optane DCPMM per socket (4 × 128 GB DIMMs on four channels per socket) in AppDirect mode, and 256 GB of DRAM (8 × 32 GB DIMMs). It is important to note that we are using a second generation DCPMM. The CPU has 16 cores (32 hyperthreads) and 48 MB of L3 cache. The server runs Ubuntu 20.04.3 LTS with kernel 5.13.0. For the workload, we use micro-benchmarks and YCSB [6] workload. Each test firstly warms up using 30 million key-value pairs [7,13,17]. Each test lasts 60 s. The total size exceeds the size of the L3-cache and can truly reflect the performance of PM. The micro-benchmarks contain the operations of lookup, insert, update, remove and scan. They are performed using 4 threads. For a fair comparison, we modified the persistence policy in other persistent indexes for the adaptation of eADR features.

4.2 Microbenchmark and Scalability Evaluation

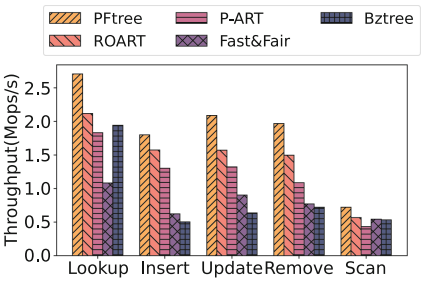

Fig. 7. Microbench (4 threads)

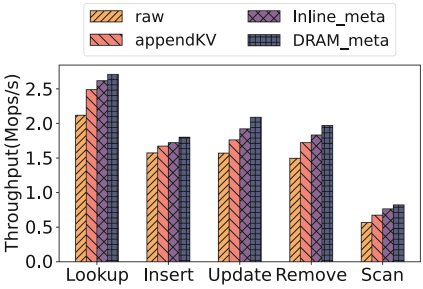

Fig. 8. Performance improvement of each optimization.

The results of micro-benchmark are in Fig. 7. The lookup performance of PFtree is 2.708 Mop/s, faster than ROART (2.118 Mop/s) and P-ART (1.832 Mop/s). This is because PFtree stores key-value pairs directly in the leaf array, whereas ROART stores pointers to leaf nodes in the leaf node array, so we need more pointer chasing in ROART. The reason why PFtree and ROART are faster than P-ART is that N16 in P-ART cannot use SIMD instructions for accelerated lookup. BzTree is fast because it used slotted-page node layout which can have good cache locality. In addition, FastFair does not use binary lookup in internal nodes, so query performance is low. PFtree also has better insertion performance (1.8 Mops/s) than the other trees. There are many reasons for this rise. (1) PFtree stores the inserted key-value pairs in the form of a leaf node array and allocates memory of multiple leaf nodes at a time, which reduces the overhead of memory allocation. (2) PFtree stores the necessary metadata and key-value together, reducing PM line accesses. And the auxiliary metadata is stored in DRAM, further reducing PM line accesses. (3) Compared with PART, the array of leaf nodes can store more key-value pairs, reducing the number of segments.

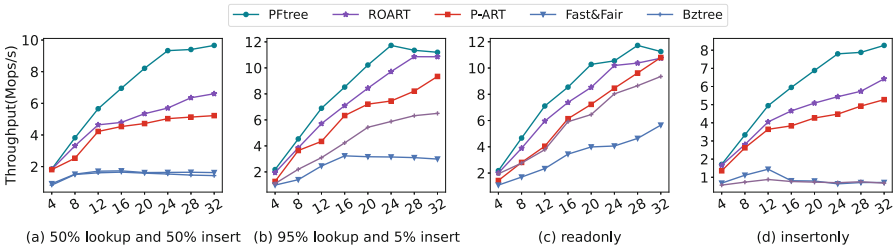

Fig. 9. The performance varying thread under YCSB workload.

For update operations, PFtree shows better for similar reasons. Key-value pairs directly stored reduces a pointer chasing and speeds up the search. More metadata stored in DRAM can help updates and they can update themselves quickly. For the remove operations, ROART needs to free the memory of the node after each deletion, while PFtree free all the memory after all key-value pairs in the leaf array are deleted.

Figure 9 shows the performance and scalability of PFtree and the state-of-the-art ART and B+tree indexes. For the write-intensive workloads A (50% lookup and 50% insert) and D (insert only), PFtree performs up to 2X better than all the other indexes; this can be attributed to PFtree storing key-value pair directly, reducing pointer jumps. And storing metadata and key-value pair closely reduces the access to PM line every time when we access the data. In addition, PFtree apply a bulk request memory policy avoids memory allocation overhead on critical paths. Other B+Tree indexes experience high latency on the critical path due to SMOs. For the read-intensive workloads B (95% lookup and 5% insert) and C (read only), PFtree outperforms all the other indexes by 1.2–7x using 32 threads. The primary reason is metadata stored in efficient DRAM speeds up the traversal and search of leaf nodes, and key-value pair stored directly reduces pointer chasing.

4.3 Factor Analysis of Each Design

Figure 8 presents the factor analysis on PFtree. We start with ROART and add proposed design features. The experiment setup is the same as in microbenchmark, using 4 threads for 60 s of test.

+ Store Key-Value Directly. We saw that this optimization worked in all of our tests. It improves lookup efficiency by 15% and delete efficiency by 11%, as well as in processes such as inserts and updates. In the lookup and scan, efficiency is improved by reducing the number of pointer chasing. In the insert and update, key-value pairs directly stored in the leaf array can not only reduce a pointer chasing, but also avoid memory allocation overhead when constructing a leaf node each time, and reduce flush times. In the remove, the original method requires the removed leaf node be reclaimed every time it is deleted, whereas in PFtree only metadata needs to be changed without memory reclamation.

+ Metadata Stores Closely with Key-Value Pair. Unlike previous data structures that store metadata in the header, PFtree stores the metadata associated with each key-value pair directly with the actual data. In this way, when accessing key-value pair each time, there is no need to read both the PM line where the metadata is located and the PM line where the key-value pair data is located as before, and fewer PM line accesses can reduce the PM read/write amplification.

+ Allocate Violate Metadata from DRAM. This is a slight improvement in all operations because each operation involves reading or writing volatile metadata, and DRAM is more efficient than PM. The improvement is even more

pronounced in insert and update operations because more metadata is written and DRAM has a higher write bandwidth than PM.

4.4 Recovery

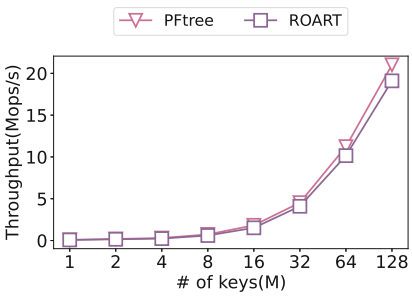

Fig. 10. Data structure recovery.

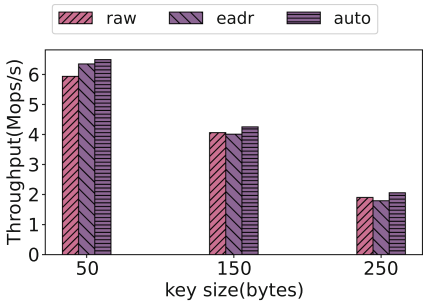

Fig. 11. Insert throughput varies with key length.

In Fig. 10, we test recovery time with different key numbers in PFtree; With 128M keys, data structure recovery takes about 21 s to reconstruct the whole metadata, which is a little slower than ROART (19 s). The main reason is that PFtree stores more volatile metadata in the leaf array to assist in search and update. During recovery, it needs to reconstruct the metadata in Fig. 4 according to the 2-byte metadata in front of each key-value pair, so it takes more time to recover.

4.5 Auto_flush Benchmark

Auto_flush benchmark focus on the persistent ways with PFtree. We have three versions of PFtree: raw-PFtree, eADR-PFtree, and auto-PFtree. Raw-PFtree persists data using the traditional flush and fence instructions. It executes the flush and fence instructions after each data writen to the PM. Based on raw-PFtree, eADR-PFtree removes flush and fence instructions directly, and takes advantage of the nature of cache persistence to ensure data consistency. Auto-PFtree is based on the discovery of the new generation of PM hardware in Sect. 3.2 and uses the adaptive persistence algorithm proposed in Sect. 3.3.

Since the length of the key and value will affect the persistent data size, and the adaptive persistence algorithm is related to the persistent data size, we explored the effects of three persistence modes under different key lengths, and the results are shown in Fig. 11. We tested insert throughput with 8 threads at key lengths of 50, 150, 250 respectively. When the key length is small, eADR mode can improve a little, and auto-PFtree mode improves less than eADR

mode. With the increase of key length, auto-PFtree insert throughput is gradually better than eADR-PFtree. This is consistent with the previous observation in Sect. 3.2. Because the persistence method used by auto-PFtree varies according to the amount of data being persisted. As the key length increases, the amount of data required for each persistent operation increases. According to the results in Sect. 3.2, when the amount of data exceeds 256B, eADR persistence cannot fully use the PM bandwidth. Flush instruction assisted persistence can obtain more bandwidth. Therefore, auto-PFtree can have large throughput when the key length is large.

Unfortunately, auto flush does not provide much performance improvement because tree traversal and lookup consume a lot of time during ART insertion, and persistence only takes a small fraction of that time, so auto flush has a limited effect.

5 Conclusion

This paper presents PFtree, a PM Line Accesses Friendly Adaptive Radix Tree optimized for eADR platform. PFtree is proposed with several optimizations. (1) PFtree store key-value pair directly in leaf arrays, reducing pointer chasing, and metadata is stored closely with key-value pair, reducing PM line accesses. (2) PFtree propose an adaptive refresh algorithm to take full advantage of the new PM hardware, based on the detailed analysis of the new generation of PM hardware.

Acknowledgements. This research is supported in part by the National Key Research and Development Program of China (2021YFC3300600).

References

1. Arulraj, J., Levandoski, J., Minhas, U.F., Larson, P.A.: BzTree: a high-performance latch-free range index for non-volatile memory. Proc. VLDB Endow. **11**(5), 553–565 (2018)
2. Chen, S., Jin, Q.: Persistent B+-trees in non-volatile main memory. Proc. VLDB Endow. **8**(7), 786–797 (2015)
3. Chen, Y., Lu, Y., Fang, K., Wang, Q., Shu, J.: uTree: a persistent B+-tree with low tail latency. Proc. VLDB Endow. **13**(12), 2634–2648 (2020)
4. 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)
5. Chen, Z., Hua, Y., Ding, B., Zuo, P.: Lock-free concurrent level hashing for persistent memory. In: 2020 USENIX Annual Technical Conference (USENIX ATC 2020), pp. 799–812 (2020)
6. 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)

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} 2018), pp. 187–200 (2018)
8. Intel: Intel optane dc persistent memory module. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html
9. Intel: Deprecating the pcommit instruction (2016). https://software.intel.com/content/www/us/en/develop/blogs/deprecat-epcommit-instruction.html
10. Intel: eADR: new opportunities for persistent memory applications (2021). https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html
11. Kim, W.H., Krishnan, R.M., Fu, X., Kashyap, S., Min, C.: PacTree: a high performance persistent range index using PAC guidelines. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pp. 424–439 (2021)
12. Krishnan, R.M., et al.: {TIPS}: making volatile index structures persistent with {DRAM-NVMM} tiering. In: 2021 USENIX Annual Technical Conference (USENIX ATC 2021), pp. 773–787 (2021)
13. Lee, S.K., Mohan, J., Kashyap, S., Kim, T., Chidambaram, V.: RECIPE: converting concurrent dram indexes to persistent-memory indexes. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 462–477 (2019)
14. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: artful indexing for main-memory databases. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 38–49. IEEE (2013)
15. Lepers, B., Balmau, O., Gupta, K., Zwaenepoel, W.: KVell: the design and implementation of a fast persistent key-value store. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 447–461 (2019)
16. Liu, J., Chen, S., Wang, L.: LB+ trees: optimizing persistent index performance on 3DXPoint memory. Proc. VLDB Endow. **13**(7), 1078–1090 (2020)
17. Liu, M., Xing, J., Chen, K., Wu, Y.: Building scalable NVM-based B+ tree with HTM. In: Proceedings of the 48th International Conference on Parallel Processing, pp. 1–10 (2019)
18. Ma, S., et al.: {ROART}: range-query optimized persistent {ART}. In: 19th {USENIX} Conference on File and Storage Technologies ({FAST} 21), pp. 1–16 (2021)
19. 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 (2016)
20. Venkataraman, S., Tolia, N., Ranganathan, P., Campbell, R.H.: Consistent and durable data structures for {Non-Volatile}{Byte-Addressable} memory. In: 9th USENIX Conference on File and Storage Technologies (FAST 2011) (2011)
21. 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 2017), pp. 349–362 (2017)
22. 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 2015), pp. 167–181 (2015)