PACTree: A High Performance Persistent Range Index Using PAC Guidelines

Wook-Hee Kim   R. Madhava Krishnan   Xinwei Fu   Sanidhya Kashyap[†]   Changwoo Min

Virginia Tech   [†]*EPFL*

Abstract

Non-Volatile Memory (NVM), which provides relatively fast and byte-addressable persistence, is now commercially available. However, we cannot equate a real NVM with a slow DRAM, as it is much more complicated than we expect. In this work, we revisit and analyze both NVM and NVM-specific persistent memory indexes. We find that there is still a lot of room for improvement if we consider NVM hardware, its software stack, persistent index design, and concurrency control. Based on our analysis, we propose ***Packed Asynchronous Concurrency (PAC)*** guidelines for designing high-performance persistent index structures. The key idea behind the guidelines is to 1) access NVM hardware in a ***packed*** manner to minimize its bandwidth utilization and 2) exploit ***asynchronous concurrency*** control to decouple the long NVM latency from the critical path of the index.

We develop ***PACTree***, a high-performance persistent range index following the ***PAC*** guidelines. PACTree is a hybrid index that employs a trie index for its internal nodes and B+-tree-like leaf nodes. The trie index structure packs partial keys in internal nodes. Moreover, we decouple the trie index and B+-tree-like leaf nodes. The decoupling allows us to prevent blocking concurrent accesses by updating internal nodes asynchronously. Our evaluation shows that PACTree outperforms state-of-the-art persistent range indexes by 7× in performance and 20× in 99.99 percentile tail latency.

CCS Concepts: • **Information systems Data structures**; **Storage class memory**.

Keywords: Non-volatile Memory, Index structures

ACM Reference Format:

Wook-Hee Kim   R. Madhava Krishnan   Xinwei Fu   Sanidhya Kashyap[†]   Changwoo Min   . 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines . In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483589

1 Introduction

The introduction of byte-addressable non-volatile memory (NVM), such as Intel's Optane DC Persistent Memory (DCP-MM), is breaking the traditional dichotomy of storage and memory. Not only that, it is fundamentally changing the way we design storage systems. For example, one of the core components of storage systems is an index structure (*e.g.*, B+-tree, hash table) [50, 71] that file systems [13, 17, 18, 32, 37, 51, 55, 60, 70, 73], key-value stores [6, 34, 42, 47, 49, 59, 72], and database systems [5, 12, 35, 54, 58, 61] use commonly.

Because of the recent commercialization of the NVM hardware, prior works either emulated using the DRAM [7, 24, 38, 53, 63, 77] or partially utilized the real hardware potential [39, 45]. Moreover, recent performance characterization studies [20, 21, 23, 31, 43, 62, 67, 68, 74] show that real NVM hardware has a lot of discrepancy in comparison with the DRAM. For example, all prior index-based works forgo the impact of non-uniform memory access (NUMA). Although the impacts of NUMA (*e.g.*, NUMA-aware writes, memory allocation) have already been studied in DRAM, this effect is more pronounced in NVM (§3.2).

In this paper, we first propose a set of guidelines for designing high-performance persistent indexes. We later use them to design a novel persistent range index that follows our design guidelines. To derive comprehensive guidelines for designing high-performance persistent indexes, we holistically analyze how current indexes work on the real NVM hardware. In particular, we investigate how both the systems software (*e.g.*, persistent memory allocator) and NVM hardware affect the design and overall performance. We conclude that designing a high-performance persistent index goes beyond just making the DRAM index crash consistent. There are several NVM unique design factors that critically impact NVM rather than DRAM.

We organize our design guidelines from our analysis in four parts: First, we present the key performance property of NVM hardware one should consider in designing a persistent index. One of our new findings is that the directory cache coherence protocol is the root cause of bandwidth meltdown in the cross-NUMA NVM access. Second, we emphasize how to

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483589

use the system component. One of our findings is that NVM memory allocation overhead is costly – much more costly than DRAM allocation – and can be the primary performance and scalability bottleneck. Third, we present several algorithmic considerations for designing a persistent index. In particular, we analyze two representative index design approaches: B+-tree and trie. We then compare how different design choices in B+-tree and trie affect a persistent index's performance and scalability. A key finding is that NVM bandwidth will be the first performance bottleneck in many cases. Lastly, we present the considerations in concurrency control. One critical finding is that the performance impact of structural modification operations (SMOs), *i.e.*, operations modifying the structure, is more significant in NVM than DRAM.

To sum up, our guidelines have two key takeaways: In a persistent index, 1) the fundamental performance-limiting factor is the limited NVM bandwidth and 2) the critical source of scalability bottleneck is the longer blocking time in critical path due to SMOs, which is further amplified by high write latency of NVM. We propose that a persistent index's design should perform *packed* NVM access to save the NVM bandwidth and exploit *asynchronous concurrency* to avoid the longer blocking time of SMOs. We call our guidelines ***Packed, Asynchronous Concurrency (PAC)*** guidelines.

Using the proposed PAC guidelines, we designed a new persistent range index ***PACTree***. PACTree is a hybrid index consisting of an optimized trie for internal nodes (search layer) and B+tree-like leaf nodes (data layer). The optimized trie consumes lower NVM bandwidth than B+-tree because it encodes partial keys in each level. Meanwhile, the B+-tree style slotted leaf nodes reduce NVM allocation overhead. Moreover, the slotted leaf nodes speed up scan operation because the sequential reads to a leaf node exploit the CPU-level and NVM-level hardware prefetcher, hiding NVM access latency. We decoupled the search layer and the data layer that allows enables PACTree to keep the cascading SMOs off the critical path *i.e.*, when the SMO of a leaf node happens, PACTree asynchronously updates the search layer to prevent costly SMOs blocking concurrent access. Unlike prior persistent indexes [8, 45, 57, 76] that place internal nodes on the DRAM to reduce the access latency, our decoupled index design allows PACTree to place even its search layer on the NVM. This placement enables a near-instant recovery as well as good capacity scaling.

PACTree supports durable linearizability [30]—a standard correctness condition for NVM data structures. We thoroughly evaluate PACTree on two NVM machines with varying NVM bandwidth. Our evaluation results show that PACTree significantly outperforms the state-of-the-art persistent range indexes in performance, scalability, and tail latency. In summary, this paper contributes the following:

- **Analysis.** We thoroughly analyze how prior persistent indexes work on real NVM hardware.

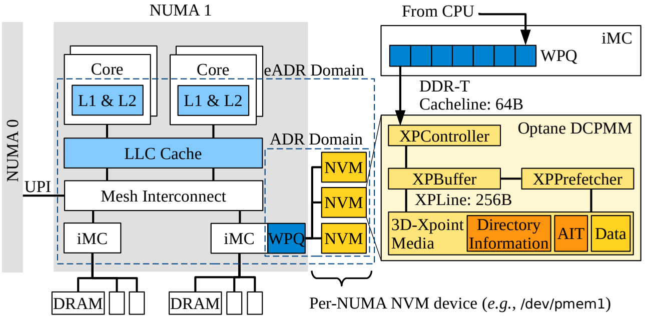

Figure 1. Architecture of byte-addressable NVM systems and the internal architecture of Intel Optane NVM (DCPMM).

- **The PAC Guidelines.** Based on the analysis, we derive 15 design guidelines, collectively named *Packed, Asynchronous Concurrency (PAC)*, for high-performance persistent index. The PAC guidelines covers four different aspects: NVM hardware, NVM software stack, index algorithm design, and concurrency control.
- **Novel Persistent Range Indexes.** We propose *PACTree*, a new Persistent range index following the PAC guidelines. PACTree is a hybrid index of an optimized trie for internal nodes and B+-tree-like leaf nodes. For the optimized trie, we propose *PDL-ART*, which is a persistent Adaptive Radix Tree (ART) supporting durable linearizability.
- **Evaluation.** Our evaluation shows that PACTree outperforms the state-of-the-art persistent range indexes by 7× in performance and 20× in 99.99 percentile tail latency.

2 Background

2.1 Non-Volatile Memory

Figure 1 shows the architecture of NVM systems and the internal architecture of Optane NVM. NVM systems support one of two *persistent domains*: (1) *Asynchronous DRAM Refresh (ADR)* mode and (2) *enhanced ADR (eADR)* mode. In the ADR mode, NVM, *write pending queue (WPQ)* in an *integrated memory controller (iMC)* are part of the persistent domain. Since the CPU caches are volatile, programs must explicitly flush cache lines using cache-line flush instructions and enforce ordering using memory fence instructions to be crash consistent. In the eADR mode, other than WPQ, the CPU caches are also part of the persistent domain, so programs do not explicitly flush cache lines. Since the eADR mode is still not available, we focus on the ADR mode.

Optane NVM internals. The CPU memory controller (iMC) communicates with the Optane NVM in the cache-line granularity (64-byte), while NVM accesses data at a coarser granularity: 256-byte (*XPLine*). Thus, a single cache-line size write can cause a read-modify-write of a 256-byte XPLine, causing write amplification generally. The Optane controller has a write-combining buffer (*XPBuffer*) and an associated XPLine prefetcher (*XPPrefetcher*). Moreover, the NVM media holds the directory information (*i.e.*, coherence states) for the current implementation of the directory coherence protocol [25].

Thus, on any coherence state modification (*e.g.*, reading a memory location from a different NUMA domain), the CPU issues *directory writes* to update the coherence state that resides on the *NVM* media. These modifications have a huge implication in terms of latency, and it is the root cause of bandwidth meltdown in cross-NUMA NVM access (§3.1.1).

Optane NVM performance. The performance characteristics of Optane NVM differ from DRAM. Not only the read and write latency are up to 2–5× higher than DRAM, but it also has 5× lower write bandwidth than DRAM [74]. NVM shows asymmetric read-write performance: write is 3–5× slower than read in latency and bandwidth [25]. In addition, sequential access is 3–5× faster than random access [25]. For writes, write buffering at 256-byte granularity improves the latency for sequential pattern; meanwhile, the random pattern leads to read-modified-write operations, which renders write combining in XPBuffer ineffective, leading to write amplification for smaller writes. Similarly, for reads, sequential patterns exploit locality and prefetchers in CPU and Optane NVM. Meanwhile, random ones avoid prefetching and require fetching 256-byte data.

2.1.1 System software stack for persistent memory

Direct Access Mode (DAX) enabled file systems (*e.g.*, ext-DAX [69], XFS-DAX [10]) mount a file system over the raw NVM device and directly expose an NVM space associated with a file to the program's virtual address space using `mmap`. NVM allocators [16, 27, 56] manage the `mmap`-ed NVM space. Unlike DRAM allocators, an NVM allocator should guarantee crash consistency to avoid its heap metadata corruption. By default, one NVM heap on one NVM raw device utilizes only NVM DIMMs and iMCs of that device, thereby underutilizing both NVM bandwidth and capacity of a system.

2.2 Persistent Range Indexes

We focus on *range indexes* that handle a large volume of data and support the scan operation. In the rest, we use *read operation* to denote `lookup` and `scan` operations and *write operation* for `insert`, `update`, and `delete` operations.

2.2.1 B+-Tree-based persistent indexes

There have been various research efforts to design efficient persistent B+-tree indexes [4, 8, 24, 45, 57, 76]. **FastFair** [24] is a lock-based B+-tree index designed to minimize crash consistency overhead by making the reader transient inconsistency tolerable. **BzTree** [4] is a lock-free B+-tree. For lock-free implementation, it relies on Persistent Multi-word Compare-And-Swap (PMwCAS) [65] primitive, which supports atomic and crash-consistent multi-word updates. BzTree keeps internal nodes immutable and performs Copy-on-Write (CoW) updates except for updating existing child pointers. **FP-Tree** [57] is a DRAM-NVM hybrid B+-tree. It places reconstructable internal nodes on faster DRAM. FP-Tree uses a hybrid concurrency control: HTM for internal nodes and spinlock for leaf nodes. It uses a key fingerprint (one byte hash of a key [57]) to accelerate a lookup operation. **LB+-tree** [45] optimizes the FP-tree design for Optane NVM. In particular, it leverages an XPLine size granularity to avoid write amplification.

2.2.2 Trie-based persistent indexes

Unlike the B+tree-based indexes, there are few trie-based persistent indexes [38, 39, 48]. **WOART** [38] is a modified ART [40] for NVM. It uses a slot array with unsorted entries to minimize the persistence cost but does not support concurrent accesses. **P-ART** [39] is the persistent version of concurrent ART [41] using RECIPE guidelines [39]. For crash consistency, P-ART re-purposes a helping mechanism to detect and fix the crash inconsistency across restarts. However, P-ART does not guarantee durable linearizability [30]. **ROART** [48] is an improved version of P-ART for supporting efficient range queries, lower memory allocation overhead, and correctness. However, since ROART inherits its rebalancing algorithm and index structures from ART [40], it still incurs high allocation overhead during SMOs, as it requires more than two leaf array allocation for every split operation.

3 Design Guidelines for Persistent Indexes

We propose guidelines for designing highly performant and scalable persistent indexes. We first extensively analyze four representative persistent range indexes – FastFair [24], BzTree [4], FP-Tree [57], and PDL-ART (our version of persistent ART guaranteeing durable linearizability described in §5.1). From the empirical analysis, we derive five findings on NVM hardware (*FH*, §3.1). We then propose ten guidelines on efficient use of system software stack (*GS*, §3.2), algorithmic considerations for persistent index (*GA*, §3.3), and concurrency control (*GC*, §3.4), making the best use of NVM hardware. We coined our findings and guidelines as ***packed, asynchronous concurrency (PAC) guidelines***.

The key PAC guidelines from a distance are that, *unlike the DRAM index, the fundamental performance-limiting factor of an NVM index is the limited bandwidth and the slow latency of NVM. We propose that an index should 1) access in a **packed** fashion to save the limited NVM bandwidth and 2) exploit **asynchronous concurrency** control to decouple the long NVM latency from the critical path.* Specifically, our PAC guidelines make several unique contributions (FH_4, FH_5, GS_2, GA_1, GA_2, GC_2) and quantify the implications of already known findings (GS_1, GA_3, GC_3).

3.1 Findings on NVM Hardware

Prior findings on NVM hardware As discussed in §2.1, *NVM bandwidth is limited (FH_1)* – a prime bottleneck in index structures. *NVM exhibits asymmetric read/write latency and bandwidth (FH_2).* Hence, for a moderate write-intensive workload, the write bandwidth becomes the first bottleneck, resulting in underutilized read bandwidth. *Sequential NVM access is preferred (FH_3).* Sequential reads can hide the slow NVM latency by leveraging both the CPU cache-line prefetcher and the NVM XPPrefetcher. Similarly, sequential writes followed by cache-line flushes efficiently use the write combining buffer (XPBuffer) in the NVM, resulting in better bandwidth utilization.

3.1.1 New findings on NVM hardware

FH_4. **NVM's data persistence cost is more expensive than we think.** In the current ADR-enabled systems with volatile CPU caches, persistence instructions (*e.g.*, `clwb`, `sfence`) are necessary to enforce the persistence and ordering of prior writes. Enforcing persistence is slow because it reveals the raw NVM media latency. Also, the current implementation of `clwb` in the current generation of Intel processors invalidates cache lines [33][1]. Thus, in current processors, persistence incurs the cost of both flushing and cache line invalidation.

FH_5. **Cache coherence protocol impedes NUMA scalability.** Due to the limited DIMM slots in a single socket system, multi-socket NUMA systems are necessary to provide larger NVM capacity and higher NVM bandwidth. However, prior research [3, 21, 74] has reported that the remote NUMA access of Optane NVM is slow. Unfortunately, there has been no in-depth analysis of the bandwidth meltdown issue in the current Optane NVM systems.

We found that *the directory coherence protocol is the root cause of the NUMA bandwidth meltdown.* This is evident from Figure 2 as the performance of the FastFair plateaus when ran using Directory coherence protocol. This is because the current Intel processor architectures rely on the directory coherence protocol among NUMA domains, and it stores the directory information on the 3D-Xpoint media. Thus, on every remote read, the coherence state change (*e.g.*, from "Exclusive" to "Shared" state with bookkeeping the remote node ID) should be updated, causing a directory "write" operation to the 3D-Xpoint media. Because of this, the remote read bandwidth is significantly lower than the remote write bandwidth. *This is detrimental because a remote read generates both read and write traffic. We verify this by running a 100% remote random reads experiment in a 64-byte granularity for an 870 MB NVM file. It generated 870 MB and 481 MB of reads and writes, respectively, when we measured using PMWatch [29] for the directory protocol.* Meanwhile, the snoop protocol [2] improves the FastFair throughput by 2.5×.

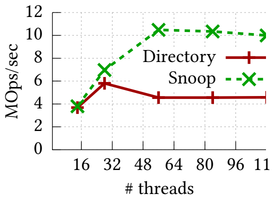

Figure 2. Performance of FastFair using snoop and directory coherence protocol for YCSB workload-A with 64M integer keys.

3.2 Guidelines for NVM Systems Software

GS_1. **Persistent memory allocation is very expensive [*FH_2*, *FH_4*].** Because NVM allocators have to provide the crash consistency of their heap metadata on failure. They rely on expensive crash consistency mechanisms. For example, the Intel PMDK allocator [27] relies on UNDO and REDO logging, which incurs six flush operations for one allocation and free pair [36]. These frequent crash consistency operations incur high overhead and generate additional NVM writes that impact the overall performance of an index [16, 44]. To quantify the overhead due to frequent memory allocation, we ran the PDL-ART using the PMDK allocator and the modified Jemalloc. Note that our modified Jemalloc allocates memory on NVM, but it does not guarantee crash consistency for NVM memory allocation and free. Figure 3 shows 2× drop in performance with the PMDK allocator due to the expensive crash consistency operations. Thus, memory allocation can severely impact performance than DRAM when designing applications for NVM.

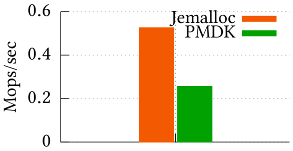

Figure 3. PDL-ART performance for insert-only workload (64M integer keys) using the non-crash consistent Jemalloc and crash cosnistent PMDK allocators.

GS_2. **Persistent memory allocation should be NUMA-aware [*FH_4*, *FH_5*].** As discussed in §2.1.1, one NVM heap by default uses NVM resources only in one NUMA domain (*i.e.*, a single-socket NVM heap). While this approach underutilizes precious NVM bandwidth, prior works have designed indexes for a single socket NVM heap [8, 9, 39, 45, 76] to avoid performance degradation due to cross-NUMA access (*FH_5*). Thus, after addressing the cross-NUMA bandwidth (*FH_5*), 1) an index should exploit all NUMA domains to maximize NVM bandwidth, and 2) the allocation should be NUMA-local to avoid NVM communication delay. Although the findings are in line with DRAM [15], we found that the impacts of such issues are more pronounced in NVM. To illustrate the importance of NVM bandwidth under-utilization, we found that FastFair is almost 2× faster on using two NUMA domains [26] than a single one with snoop coherence protocol.

3.3 Guidelines for Persistent Index Algorithm

GA_1. **Lookup operation should consume minimal NVM bandwidth [*FH_1*, *FH_2*].** The `lookup` operation is the most common operation (§2.2). Moreover, it is critical for high performance because of NVM read bandwidth utilization. We can estimate the NVM bandwidth consumption of a `lookup` operation as the total NVM reads for total key comparisons until reaching the target key. Our worst-case analytical model for NVM bandwidth consumption of a B+tree (BW_{BTree}) and a trie (BW_{Trie}) is as follows:

[1] We verify this by continuously using `clwb` and read some cache lines.

[2] Some servers provide options for cache coherence protocols in the BIOS setting.

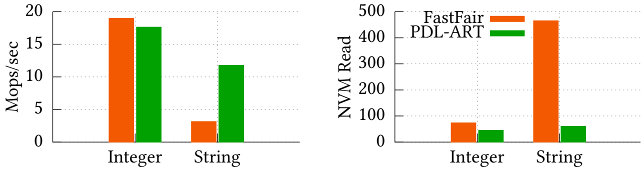

Figure 4. The performance and total NVM read (GB) of FastFair and PDL-ART for 64M lookups for integer and string keys.

$$BW_{Btree} = \lceil \log_F K \rceil \cdot \log_2 F \cdot S \quad (1)$$

$$BW_{Trie} = \log_2 F \cdot S + S \quad (2)$$

where F, K, and S are fan-out of a node, the number of key-value pairs in an index, and a search key length, respectively.

A B+tree performs a binary search in a node ($\log_2 F$) with a full key comparison (S) for each level of a tree ($\lceil \log_F K \rceil$). A trie performs a binary search of a partial key in a node ($\log_2 F$) at each level. Since internal nodes of a trie encode a key, its height cannot exceed the search key length (S), and one full key comparison (S) is needed at the end. The above equations show that a B+-tree consumes more NVM bandwidth than a trie. For instance, a trie (F = 256) consumes 3.8× less NVM IO for a lookup operation than a B+tree (F = 32) for 100M key-value pairs of the 8-byte key. However, the actual NVM IO would be different due to other architectural components, including a larger access granularity (*e.g.*, 256-byte XPLine), hardware prefetcher, and CPU cache. To check our model is useful in a real NVM, we ran a 100% lookup workload (YCSB C) such that 28 threads concurrently perform 64M lookup operations. We used two data sets with 64M key-value pairs: 8-byte integer key and 23-byte string key. Figure 4 shows that FastFair (B+tree) does more NVM reads (7.7×) as the key is longer (string key), and PDL-ART, which uses trie, has 3.7× higher throughput.

GA_2. **Read operation should minimize NVM writes [FH_1, GS_1].** Lookup and scan operations can incur additional NVM writes. For instance, pessimistic concurrency control mechanisms such as mutex, spinlock, or readers-writer lock incur NVM writes when a reader modifies the lock status. Since NVM write bandwidth is scarce, these additional writes impact the read scalability and performance. For example, we observed that the read-only workload (YCSB Workload C: 64 M lookups using 28 threads against 64 M key-value pairs of 8 B keys) generated an additional 1.4 GB of NVM writes in FastFair, which ideally should be close to zero. Placing the lock on the DRAM may be an option, but it leads to additional pointer chasing while checking the lock status, and such a design is not prefetcher-friendly.

GA_3. **Write operation should minimize NVM memory allocation [GS_1, FH_2, FH_3].** Frequent NVM allocation/free operations can impede the scalability and performance of an index because NVM allocators 1) incur high crash consistency overhead, 2) consume the NVM write bandwidth, and 3) are often a scalability bottleneck. Hence, a persistent index should minimize NVM allocation/free in its write operations.

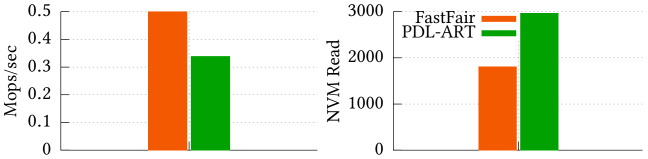

Figure 5. The performance and total NVM read (GB) of FastFair and PDL-ART for 64M scan operations on 64M integer keys.

To see the performance impact of NVM allocation/free, we ran FastFair, BzTree, PDL-ART for an insert-only workload (YCSB Load A: 64M insert of 8B keys and 8B values using 28 concurrent threads on a server configured with the snoop coherence protocol). We measured the overhead of the PMDK allocator using Linux `perf` [2]. FastFair, PDL-ART, and BzTree spent 2%, 20%, and 40% of their time in the PMDK allocator, respectively. As a result, FastFair outperforms BzTree and PDL-ART by 3× and 1.2×, respectively.

FastFair embeds 30 8B-key and 8B-value pairs in a node. It needs NVM allocation/free only when a node split/merge occurs. Thus, it incurs the lowest NVM allocation overhead. Unlike FastFair, PDL-ART does not embed a key-value pair in a leaf node. Hence, it needs one NVM allocation for a key-value pair on every insert. BzTree modifies an internal node using the CoW technique, which incurs an NVM allocation.

GA_4. **Write operation should minimize NVM persistence operation [FH_4, GS_1].** For a high performant index, an efficient crash consistency mechanism is critical to minimize additional NVM writes and expensive persistence operations (`clwb` and `sfence`). For instance, any data (*e.g.*, permutation array in FPTree [57]) that are not mandatory for correct recovery are unnecessary to persist. Thus, minimizing persistence operations not only saves NVM write bandwidth but also minimizes the critical section latency.

GA_5. **Scan operation should maximize sequential read to NVM [FH_3].** Since sequential reads in NVM are faster than random reads, they directly affect the performance of scan operations. Therefore, for fast scan operation, a node structure in an index should be prefetcher friendly (*i.e.*, sequential read) and cache line efficient. We ran 64M scan operations on 64M 8-byte integer keys with 28 concurrent threads to see how a node structure affects the scan performance. We compared the performance and generated NVM read (GB) of FastFair and PDL-ART in Figure 5. FastFair embeds sorted key-value pairs in a leaf node. This enables its scan operation to exploit the fast NVM sequential reads. However, PDL-ART stores key-value pairs outside of a leaf node, requiring multiple random NVM reads. As a result, FastFair outperforms PDL-ART by 1.5× with 1.6× less reads.

3.4 Design Guidelines for Concurrency Control

GC_1. **Maximize concurrent access for both scalability and full NVM bandwidth utilization.** Single-threaded access cannot fully utilize the NVM bandwidth. A prior work reports [74] that we need eight concurrent writers and 20 or

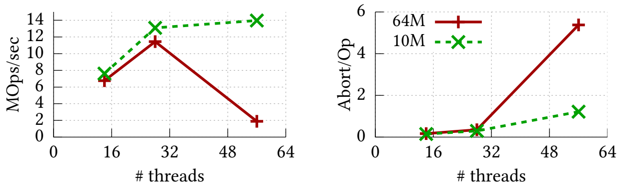

Figure 6. Performance and the number HTM aborts per operation in FP-Tree (50% lookup and 50% insert operations).

more concurrent readers to saturate the write and read bandwidth, respectively. Thus, an optimized concurrency control scheme of an index should fully exploit the available NVM bandwidth by concurrently allowing readers and writers.

GC_2**. Minimize the blocking time of structural modification operations.** Insert or delete of a key can trigger the structural modification operations (SMO) of an index (*e.g.*, split/merge of nodes in B+-tree). Moreover, it can trigger cascading structural modifications to ancestor nodes. An SMO (especially when cascaded) affects scalability in many indexes because it is in the critical path (*e.g.*, insert or delete of a key) and blocks other concurrent threads (or makes them retry in a lock-free design). This performance degradation is more severe in persistent indexes. The NVM write and persistence operations (triggered by SMOs) are high latency operations that lengthen the critical section, stalling other concurrent threads. Prior studies [1, 8, 45, 57, 76] address this issue by placing internal nodes in DRAM. Such a design can reduce the persistence overhead. However, SMOs still block concurrent threads and also increase the recovery time as the internal nodes have to be rebuilt at every startup. An ideal approach is to decouple SMOs off the critical path that reduces the blocking time. Moreover, we can store internal nodes on NVM to further reduce recovery time.

GC_3**. An index's data size should not affect the operation progress.** Large memory capacity is one of the main benefits of using NVM. Therefore, a persistent index should be designed to deal with a large data set and guarantee the progress of an operation with a large data set. Persistent indexes (*e.g.*, FP-tree, LB+-tree) relying on HTM could suffer from dealing with a large data set. Because current Intel HTM can only support L1 cache size data, HTM could abort, especially with a large data set due to the capacity miss.

To see the impact of HTM aborts on performance, we ran FP-tree for 10M and 64M uniform random integer keys. We ran a 50% lookup and 50% insert workload on a server configured with the snoop coherence protocol. Figure 6 shows that HTM aborts significantly increase with larger data sets and more concurrent threads, substantially degrading the performance. In the case of 56 threads and 64M keys, on average, 5.4 aborts happened for every index operation.

3.5 Discussion

PAC guidelines for eADR mode. We expect that the PAC guidelines are still useful in the eADR mode, in which CPU caches are persistent. Although explicit persistence instructions, such as flushes and fences, are not necessary for the persistent cache, we expect that the bandwidth of NVM will remain the main performance bottleneck. In addition, the crash consistency (for multi-pointer updates) and scalability limitations are relevant in the eADR mode. Therefore, we expect that the PAC guidelines and PACTree design for hardware (FH_5), bandwidth conservation (GA_1-GA_3, GS_1-GS_2), and concurrency (GC_1-GC_3) are applicable to the eADR mode.

Applying PAC guidelines beyond persistent indexes. We can apply PAC guidelines to other NVM software designs beyond PACTree and persistent indexes. For instance, our hardware findings (FH_1-FH_5), guidelines to conserve bandwidth (GA_1-GA_4), and exploit concurrency (GC_1-GC_3) can be applied to the design of NVM file systems. Also, PAC guidelines can improve the in-memory storage systems that use persistent memory as a large volatile memory. Since persistent memory does not provide persistence, in-memory storage systems cannot leverage design guidelines for persistence. However, other guidelines such as guidelines for NUMA awareness (GS_2), conserving NVM bandwidth(GA_1-GA_4), and concurrency(GC_1-GC_3) are valid even if persistence is not provided.

4 PACTree Overview

Following the PAC guidelines, we propose PACTree, a persistent and hybrid range index consisting of trie-based internal nodes (search layer) and B+Tree style leaf nodes (data layer). In PACTree, we decouple the search layer and the data layer—a key design aspect to fully exploit the PAC guidelines. This design enables PACTree to use hybrid indexes, a critical aspect to perform packed NVM access (§4.2). Furthermore, the decoupled design exploits asynchronous concurrency to reduce the blocking time due to the SMOs (§4.3) and consequently achieves high scalability. PACTree uses an optimized persistent version of ART, called PDL-ART (§5.1), as its search layer and slotted leaf nodes (§5.2) as its data layer. We designed PDL-ART by optimizing the volatile ART for NVM, particularly by adding crash consistency and guaranteeing durable linearizability. The following section provides a short background on the ART index and its concurrency control mechanism, ROWEX.

4.1 ART Primer

ART [40, 41] is a variant of the radix trie, which is optimized for reducing the space overhead. The radix trie stores partial keys on its node, which incurs a lot of pointer chasing overhead and space under-utilization. To improve space utilization, ART adaptively expands or shrinks the node sizes by the number of entries that are stored in the node. Also, it employs path compression to reduce the pointer chasing overhead, which further improves space utilization.

ART employs ROWEX (Read-Optimized Write-Exclusive) [41], which is an optimized synchronization technique that

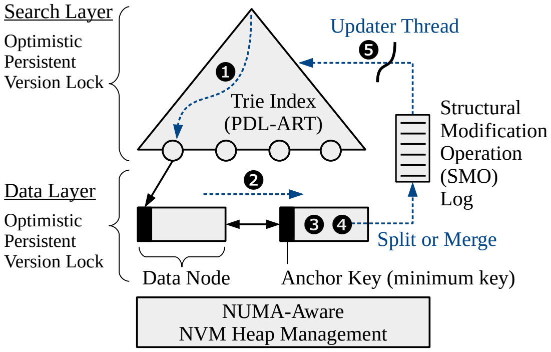

Figure 7. Overall architecture of PACTree.

guarantees exclusive writes and non-blocking reads. In ROWEX, the per-node lock is acquired by the writer before modifying an ART node, and updates to the node are always atomic. This enables readers to see the consistent status of a node and proceed without blocking.

4.2 Index Architecture

The data layer in PACTree stores the key-value pairs in a data node, and the search layer stores partial keys in a trie to locate the date node. The *Anchor key* is the smallest key of a data node when the data node is created. It never changes after creating the data node. Each data node keeps a specific range of keys between two adjacent anchor keys. If a node split (merge) happens, only the latter half of the current (right) node is moved to the new node, thereby keeping the anchor key of a node intact.

Search layer [GA_1]. Our trie-based search layer indexes anchor keys for every data node. The use of a trie-based search layer complies with GA_1, as it consumes significantly less NVM bandwidth (see Figure 4). Unlike the B+Tree, an internal node of trie keeps only the partial key, which consumes less space, and more importantly, it incurs only partial key comparison while traversing internal nodes. This is critical to efficiently consume the NVM read bandwidth, given that the NVM bandwidth becomes the first performance limiting factor and consequently achieves better performance.

Data layer [GA_3, GA_5]. PACTree's data layer is a doubly linked list of B+-tree-like leaf nodes, called *data nodes*, each containing multiple key-value pairs (see Figure 8). The slotted list structure enables insert, delete, or update operations to avoid the expensive persistent memory allocation (GA_3) and the scan operation to benefit from the sequential and localized NVM access (GA_5). Moreover, we adopt two B+-tree optimization techniques: *key fingerprinting* for fast lookup and *permutation array*[3] for efficient scan operations.

4.3 Concurrency Control

Read-optimized concurrency control [GC_1, GC_3, GA_2]. For scalability, PACTree maximizes the concurrent execution by using the read-optimized concurrency control schemes (GC_1). Both the data layer and the search layer relies on *optimistic version lock*. The lock provides exclusive access to a writer and concurrent access to readers. A reader optimistically performs a read on a data node. It checks the version number before and after the read operation. If there is a mismatch or the version number is odd, the reader retries. The read-optimized scheme of the optimistic version lock provides two properties: (1) readers do not change the lock status on NVM; hence, no writes (GA_2); (2) unlike HTM based approaches [45, 57], the progress in our lock-based approach is not affected by data size/footprint (GC_3).

Asynchronous structural modification [GC_2, GA_3, GA_4]. Insert and delete operations can trigger a structural modification. When there is no empty key-value slot for an insert, PACTree splits the data node to secure new empty slots. We add a new data node to the search layer for future access. A delete operation can trigger the merging of two data nodes and deleting the old data node from the search layer.

To avoid SMO being a scalability bottleneck, we propose an *asynchronous structural modification* approach, which decouples the expensive search layer update from the critical path. As illustrated in Figure 7, once a data node is split (or merged), PACTree logs which nodes are split (or merged) to the SMO log *without* updating the search layer (❹). A background updater thread replays each SMO log entry to either add a new data node to the search layer or remove an obsolete one from the search layer (❺).

However, PACTree should correctly handle the inconsistency between two layers when the search layer is not synchronized with the data layer. For example, a newly split node is not reachable from the search layer until the search layer is updated. To handle such inconsistency, we propose an *ephemeral inconsistency tolerable design* to guarantee correct operation at all times. The unsynchronized search layer will lead to a wrong yet adjacent data node (❶). Even if that happens, PACTree can still locate a correct data node by traversing the data layer, which is a doubly linked list, comparing the node's anchor key and a search key for operation (❷). Once locating a proper data node, PACTree performs the operation as usual (❸❹). Note that our evaluation shows the window of such inconsistency is very brief, even in the worst case workload (see §6.8). Note that *inconsistency tolerable design* in FastFair [24] is fundamentally different from PACTree. FastFair proposes the inconsistency tolerable design to eliminate the crash consistency overhead. However, its SMOs are still synchronous and are always on the critical path. On the other hand, PACTree's *ephemeral inconsistency tolerable design* of the SMO is designed for the asynchronous update of the search layer for maximizing the concurrency.

4.4 Crash Consistency

Mostly log-free crash consistency [GA_4]. PACTree guarantees crash consistency and prevents persistent memory

[3] An indirection array for sorted key access in a data node [49].

leaks upon unexpected crashes (*e.g.*, software crashes, sudden power off). PACTree does not rely on expensive logging techniques not requiring structural modification. Instead, it leverages the slotted list and the valid bitmap in a data node. The valid bitmap works as a durability point for the common case writes (§5.5), while the SMO log guarantees crash consistency and prevents persistent memory leaks for the writes that cause a split/merge (§5.6).

Selective persistence in a data node [FH_4]. Not all information in a data node must be persisted for crash consistency. Specifically, a permutation array stores the indices of keys of a data node in a sorted manner for faster scan operation. It can be recomputed as long as PACTree guarantees consistency of key-value pairs in a data node. PACTree does not guarantee the consistency of the permutation array to avoid persistence overhead and cache-line invalidation. Instead, it regenerates the permutation array on demand if necessary.

4.5 Persistent Memory Management

NUMA-aware memory management [GS_2, FH_5]. To fully utilize the NVM bandwidth and capacity in a multi-socket NUMA machine, PACTree manages the per-NUMA NVM heap (GS_2). PACTree allocates persistent memory from a NUMA-local NVM heap that avoids the costly cross-NUMA NVM accesses for subsequent writes.

5 PACTree Design

5.1 Search Layer: PDL-ART

We choose the concurrent ART [41] for the search layer due to its space efficient packed access and good scalability. There are a few persistent ART variants, but they have critical limitations, as discussed in §2.2.2. To overcome those limitations, we propose *Persistent Durable Linearizable ART (PDL-ART)*. Besides porting the volatile memory allocation to NVM, we made the following design changes to the volatile ART.

(1) Durable linearizability and recovery. Since ART supports non-blocking reads (§4.1), readers can read non-persisted writes. This violates the durable linearizability guarantee; *i.e.*, keys read before a crash are not guaranteed to exist post the crash recovery. For recovery, using the ROWEX for concurrency control (§4.1) requires resetting the per-node lock status by visiting every ART node across crashes, which is expensive. To overcome these issues, we employ an *optimistic version lock* with a *global generation ID*.Now with an optimistic version lock, a reader cannot access the node until the writer releases the lock after persisting the update operation. Thus, by blocking reads, we guarantee durable linearizability. This design choice is better in the context of PACTree because we use PDL-ART as the search layer that is rarely updated. Furthermore, we reset the per-node lock states for the correct recovery with the help of a global generation ID, which we increment whenever we start the PDL-ART instance. Since this ID is encoded in the version lock state, the previous lock state automatically becomes void (unlocked) after incrementing the global generation ID.

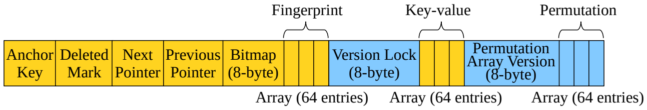

Figure 8. Data node layout. Blue one will not be persisted.

(2) Log-free crash consistency. We propose a log-free crash consistency by leveraging the concurrency control property of the PDL-ART. The use of version lock in the PDL-ART gives exclusive write access, and therefore crash consistency for the writes can simply be guaranteed by persisting the store instructions in the exact same order as the program execution order; *i.e.*, if the store persistence order is the same as the execution order, there can not be any inconsistency in the event of a crash. We need to consider two cases to persist store instructions in their execution order.

Stores modifying only a single cache line. In this case, the persistence order is equivalent to the program execution order wherever a crash happens. That is because x86 architecture never reorders store instructions, and a cache line is a persistence unit to WPQ in Optane NVM (see Figure 1).

Stores modifying multiple cache lines. In this case, we first persist all stores, except for the node metadata. We then write and immediately persist the metadata. This guarantees crash consistency because the metadata update in ART is a linearization point when the change becomes visible. Hence, ART becomes crash consistent by `clwb`/`sfence` before updating and persisting the metadata.

(3) Persistent memory leak prevention. ART allocates a new node, initializes it, and atomically links it to the parent node making the new node visible. If a crash happens before persistent linking, the new node will be unreachable, resulting in a persistent memory leak. To prevent this, PDL-ART manages small size allocation logs on NVM to keep the newly allocated yet not persistently linked nodes. To make NVM allocation and logging atomic, we rely on the *malloc-to* semantics of the NVM allocator, which allocates on the NVM and persistently attaches it to the specified address atomically. Upon recovery, PDL-ART checks all addresses in the allocation logs whether it is reachable or not. The non-reachable addresses are freed to avoid memory leaks.

5.2 Data Layer

The data layer is a doubly linked list of nodes, as shown in Figure 8. It holds unsorted key-value pairs. Moreover, it maintains a fingerprint array for fast lookup, and a permutation array for the fast scan. It also maintains an 8-byte bitmap for marking the validity of a key-value slot. Thus, updating the bitmap using an 8-byte atomic write is a linearization point for one or more key changes in a data node. Each node has an 8-byte version lock with a 4-byte generation ID and a 4-byte version number (§5.7). PACTree reconstructs permutation array across reboots by sorting keys in a data node instead of persisting it as a common case optimization. We ensure the validity of the permutation array by comparing its version with that of the data node before every scan operation. We set the number of entries to 64 on purpose for several reasons: (1) aligning the fingerprint and permutation array to a cache line, (2) atomically updating an 8-byte (64-bit) bitmap representing all entries in a node, and (3) accelerating the 64-byte fingerprint matching using a single SIMD (AVX512) instruction. For variable length string key, PACTree stores a maximum of 32-byte key and an 8-byte value in a node. For larger key and value size, we store the partial key in a node then store the rest of the key and values out of the node.

5.3 Lookup Operation

We perform a lookup operation in two phases: (1) traverse the search layer to get to the data layer (2) and finally traverse the data layer to get the target key.

(1) Traversing the search layer. A lookup operation first traverses the search layer by comparing the target key with the partial anchor keys of data nodes. If the search layer is not synchronized with the data layer, the traversal will reach the adjacent node, called the *jump node*. Otherwise, it will directly reach the *target node*.

(2) Traversing the data layer. Due to the asynchronous update of the search layer, the jump node is not guaranteed to be the target node where the key resides, so PACTree first checks the key range of the jump node using the anchor key. If the target key is smaller than the anchor key of the jump node, it traverses to the left neighbor data node, and else it traverses to the right. The target key within the key range means that the search layer traversal has reached directly to the target data node, which is the common case, as we illustrate in §6.8. On reaching the target node, PACTree first matches the fingerprint and then compares the full key only for those keys whose fingerprint matched. The fingerprint matching reduces a full key comparison and improves the lookup performance. After finding the key, it checks the version number of the node after reaching the node. If it is changed, PACTree restarts the data layer traversal because of a concurrent write.

5.4 Scan Operation

The scan operation first finds the target data node of the minimum key of a scan range. The finding target data node is the same as the lookup operation as we described in the previous section. After finding the target node, the scan operation first checks whether it needs to reconstruct the permutation array. If the node version and the permutation array version are different, it first reconstructs the permutation array. Then it traverses the key-value array in the order of permutation array. When the given range of the scan operation is bigger than the number of the key-value pairs in a single data node, the scan operation traverses to its right neighbor data node. While traversing the neighborhood, it acquires/releases the optimistic version lock for each node.

5.5 Insert/Delete/Update Operations

All write operations first lookup the target node and then acquire the version lock of the node, perform the critical work, and finally release it.

Insert. PACTree inserts the key and value to the first empty slot by checking the bitmap. Then it adds the fingerprint of the key to the corresponding position in the fingerprint array. After persisting the key-value and fingerprint, PACTree atomically updates the bitmap slot to one.

Delete. Once PACTree locates the key in the target node, it atomically resets the corresponding bit in the bitmap.

Update. PACTree first adds the key and updated value to an empty slot and updates the fingerprint array accordingly. Then it persists the added key-value pair and fingerprint. Finally, it atomically updates the bitmap by resetting the old key-value pair but setting the new key-value pair to one.

Crash consistency of a data node. The crash consistency protocol of a data node is simple. All the changes except for the bitmap should be persisted first (*i.e.*, durability point) then the bitmap is correspondingly updated using an 8-byte atomic write (*i.e.*, linearization point). Finally, PACTree releases the lock after persisting the bitmap. Upon recovery, the bitmap is the source of truth; all the bits that are set in the bitmap array are valid keys, while the rest are ignored.

5.6 Split and Merge of a Data Node

The split and merge operations are similar to that of the traditional B+-tree. However, our data layer is a doubly-linked list of data nodes, so there is no cascading split and merge operations in the data layer. Instead, the parent entry of the newly created data node or deleted data node will be updated to the search layer.

Split in the data layer. Split is triggered when attempting to insert a key-value pair to a full node. Once the node lock is acquired, a writer logs the split information in a *per-thread SMO log*. Once the SMO log entry is persisted, the writer allocates a new data node. The persistent pointer of a new node is atomically persisted at the placeholder in the SMO log entry by the NVM allocator to avoid memory leaks. PACTree copies the larger half of key-value pairs to the new data node with associated metadata. And it links the left and right of the new node to the respective left and right node of the splitting node. After the new node is persisted, it is linked to the right of the splitting node. The copied key-value pairs are deleted from the splitting node by atomically updating the bitmap, and then PACTree persists the splitting node. Finally, the new node is linked to the left of the splitting node's right node and, PACTree persists the updated pointer.

Merge in the data layer. When a delete operation causes the number of keys in two adjacent data nodes to be less than half of the key array capacity (32), PACTree merges the right data node to the left data node where a key is being deleted. PACTree first acquires the locks of two nodes, which will be merged, and then logs the merge information to the per-thread SMO log. After persisting the SMO log entry, PACTree moves the key-value pairs in the right node to the left and updates the fingerprint and bitmap in the left node. After persisting the left node, PACTree logically deletes the right node by marking the deleted flag in the node. Finally, it updates the sibling pointers.

Asynchronous update of the search layer. A background updater thread replays SMO log entries to synchronize the search layer. It merges and sorts per-thread SMO log entries in timestamp order when the log entry is inserted. Then it replays the entries in timestamp order, inserting the split node's anchor key and deleting the merged node's anchor key from/to the search layer. The updater thread also physically deletes the merged node that has been logically deleted. We use Epoch Based Memory Reclamation (EBMR) techniques [19, 22, 52] to ensure that no thread is reading the merged node. An epoch is defined as all threads finish their current operation. PACTree should wait for two epochs once the merged node is deleted from the search layer. The first epoch ensures that there is no new reference from the search layer. The second epoch ensures that all new references starting from the previous epoch finish their access. Thus, after two epochs, the merged node is guaranteed to be inaccessible, and it can be safely freed.

5.7 Optimistic Persistent Version Lock

We propose an *optimistic persistent version lock* for node level locking in search and data layer. The optimistic persistent version lock is an 8-byte version composed of a 4-byte generation ID and a 4-byte version number. It basically follows the typical version lock protocol. A writer atomically increments the version upon lock and unlock. When a version is an odd number (*i.e.*, the write lock is held), a thread should wait until the version becomes an even number (*i.e.*, the write lock is released). A reader first checks if there is no writer (*i.e.*, an even version number). If so, it optimistically performs an operation and then checks the version again. If two versions vary, the operation is retried. Version lock is a good fit for NVM because it provides high concurrency, and also readers do not change the lock state, which saves NVM bandwidth.

The global generation ID is monotonically increased when PACTree is loaded. When the lock is updated, the global generation ID is associated with the version number. When accessing the version lock, it first checks the associated generation ID and the global generation ID. Mismatched generation ID denotes that the lock has already expired, so a thread atomically initializes the lock state to proceed. Incrementing the global generation ID resets all locks at once.

5.8 NUMA-Aware Persistent Memory Management

Per-NUMA NVM pools. PACTree creates a separate NVM pool for search layer, data layer, and logs to enhance the spatial locality of NVM data. Each NVM pool consists of a per-NUMA sub-pool. PACTree always allocates NVM memory from a NUMA-local pool to reduce the cross-NUMA traffic.

Compact persistent pointer representation. PACTree divides a 64-bit pointer into two parts: The upper 16 bits store an NVM pool ID, and the lower 48 bits stores an offset from the NVM pool. We also allocate a base address pool array to store the base addresses of NVM pools, which is initialized on loading each NVM pool. PACTree gets a raw pointer for actual NVM access by adding the base address of an NVM heap to the pointer's offset.

5.9 Recovery

When there are any remaining SMO log entries upon start, PACTree starts the recovery procedure for SMO.

Recovery of a split operation. PACTree first checks whether the SMO log entry of the new node address is `NULL`. If `NULL`, the split is interrupted before a new node allocation. Hence, PACTree re-executes the split operation based on the information in the log entry. If the new node is already allocated (*i.e.*, non-`NULL`), PACTree checks whether the new node is fully initialized by checking if the splitting node points to the new node according to the persistence ordering (§5.6). If the new node is not fully initialized, PACTree re-initializes the new node. At this point, the key-value pairs and sibling pointers of the splitting node have not been changed (while the bitmap may be updated). Thus, PACTree performs the same new node initialization steps described in §5.6. Once it is confirmed that the new node is fully initialized, PACTree checks if the new node is fully inserted into the list in the data layer. At this point, the sibling pointers of the new node are guaranteed to be correct. Thus, PACTree checks if the left and right nodes from the new node correctly point to the new node. If not, PACTree corrects the sibling pointers of the new node's neighbor nodes. Now the recovery of the data layer is done. After that, PACTree checks whether the parent entry of the new node exists in the search layer. If it does not exist, PACTree inserts the parent entry to the search layer and then deletes the SMO log entry.

Recovery of a merge operation PACTree first checks if the SMO log entry of the deleting node address is `NULL`. If so, it means the node is already merged and deleted. Otherwise, it resumes the merging of two nodes. Since PACTree does not modify anything on the deleting node, the information on the deleting node at this point is guaranteed to be correct. Further, it checks whether the key-value pairs on the deleting node are already copied to the left node. Else, it copies the keys on the deleting node to the left following the steps described in §5.6. At this point, two nodes are merged. Now, PACTree checks if the deleting node is unlinked from the data layer. It obtains the left and right nodes of the deleting node from its sibling pointers, which never changes during the merge operation. Then it checks whether left and right nodes are still pointing to the deleting node. If so, it fixes those pointers to remove the deleting node from the list, which marks the completion of merging at the data layer. PACTree checks if the search layer still indexes the deleting node. If so, it deletes the deleting node from the search layer. Finally, it frees the deleting node and the log entry.

6 Evaluation

Evaluation environment. We use a two-socket DCPMM machine with Intel Xeon Platinum 8280 processors (28 physical/56 logical cores) per socket, 3.0 TB of DCPMM (1.5TB per socket), and 768 GB of DRAM. We used gcc 8.4.0 with `-O3` optimization. We set the snoop coherence protocol and `pmempoolset` [26] to exploit NVMs on all NUMA domains.

Workload configuration. We used the index-microbench [14, 66] that generates YCSB [14] workload. We use both the integer (8-byte) and string keys (23-byte on average) with uniform and Zipfian distribution. The value size is 8-byte, which is the default setting of the index-microbench and is used in previous studies [7, 24, 38, 57, 75]. We do 64M operations after populating the indexes using 64M keys except for the LOAD A evaluation. Note that since most of the prior indexes do not support the update operation, we replace the update operation to insert operation similar to the previous work [39]. We include the evaluation results for only the Zipfian distribution due to the space limitation. The performance trends are similar for both the uniform and Zipfian distributions.

Target comparisons. We evaluated and compared PACTree with all the available and usable state-of-the-art representative indexes. For FPTree [57] evaluation, we got the latest binary from the authors, which is more optimized than the original one. We did not include LB+-tree [45] because its open-sourced version is unstable due to HTM aborts, and it does not implement a software fallback mechanism. Both the LB+Tree [45] and uTree [8] are variants of hybrid DRAM+PM B+-tree indexes, so FPTree is a representative index for such hybrid indexes. We also evaluated the other state-of-the-art B+tree and trie indexes such as BzTree [4] (lock-free B+tree), FastFair [24] (logless crash consistency), and PDL-ART. We do not present string keys numbers for FPTree because the author's provided binary does not support variable-length keys. For a fair comparison, we ported all the indexes to use the PMDK allocator [28].

6.1 Performance and Scalability Evaluation

PACTree vs B+Tree indexes. Figure 9 and Figure 10 show the performance and scalability of PACTree and the state-of-the-art B+tree indexes. For the write-intensive workloads (W-A, L-A), PACTree performs up to 4× better than all the other B+tree indexes; this can be attributed to the asynchronous search layer update in PACTree while the other B+Tree indexes experience high latency in the critical path due to the SMOs. For the read-intensive workloads (W-B, W-C, W-E), PACTree outperforms all the other B+tree indexes by up to 3.2×. The primary reason is the trie-based search layer in the PACTree which incurs only a partial key comparison to get to the target node. The B+tree indexes require full key comparisons, which not only increase the read latency but also consume more NVM bandwidth which ultimately becomes a performance bottleneck. Apart from the aforementioned benefits, PACTree has other design benefits over the existing B+tree indexes, which we discuss below.

PACTree vs BZTree. For write-intensive workloads, the main performance bottleneck in BZTree comes from a large number of NVM memory allocations and a high number of `clwb`/`sfence` incurred per insert operation due to the use of PMWCAS. Further, BZTree spends about 40% of its time in memory allocation, and in total, it requires at least 15 flushes per insert operation. Alternatively, with the slotted leaf node, memory allocation overhead in PACTree can be amortized, and with the asynchronous search layer update, the number of `clwb`/`sfence` in the critical path is significantly reduced. Also, with slotted leaf nodes, PACTree can exploit the fast sequential reads for the range scan while BZTree experiences overhead mainly from additional dereferencing and snapshotting for every scan operation.

PACTree vs FPTree. The performance of FPTree slumps for higher thread counts (except for W-C) due to HTM aborts. Even for lower thread counts (< 32), PACTree outperforms FPTree by up to 1.5× across all workloads. Although FPTree stores its internal nodes on the DRAM, still the SMOs happen in the critical path but, in PACTree, though the internal nodes are on the NVM, the SMOs are off the critical path and consequently have a better write performance than FPTree. For instance, FPTree performs on-par with PACTree for read-only workload C, but even for a smaller write ratio (*e.g.*, W-B), FPTree experiences a dip in performance due to the SMOs in the critical path.

PACTree vs FastFair. An interesting performance trend in FastFair is that its performance drops up to 3× for string keys than that of the integer keys. This is because the key-value pairs are embedded in the leaf node for the integer keys, which makes FastFair cache line efficient. Meanwhile, for string keys, the leaf node contains only pointers to the keys, which incurs additional pointer chasing and consequently a poor performance. Alternatively, PACTree shows a similar performance trend for both the string and integer type keys.

PACTree vs PDL-ART. PACTree performs up to 3× better than PDL-ART across all the workloads. Similar to BZTree, PDL-ART is NVM allocation heavy; *i.e.*, each insert operation incurs an NVM allocation, and hence it is ~3× slower than PACTree for write-intensive workloads. Also, in PDL-ART,

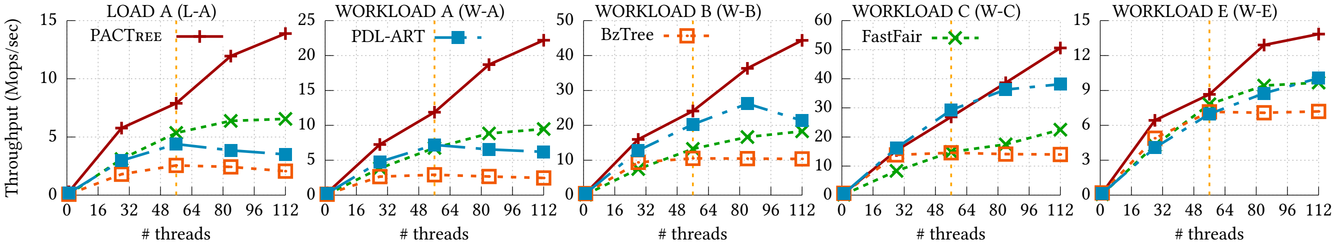

Figure 9. Performance comparison of persistent indexes for YCSB workloads of string keys with Zipfian distribution.

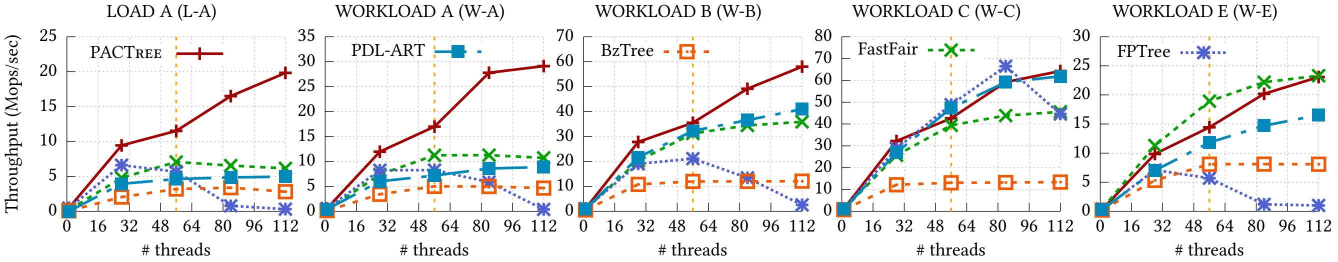

Figure 10. Performance comparison of persistent indexes for YCSB workloads of integer keys with Zipfian distribution.

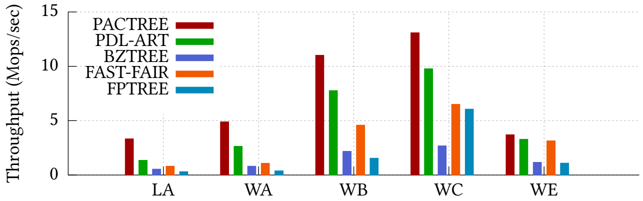

Figure 11. Performance comparison of persistent indexes for uniform YCSB workloads on a low bandwidth NVM machine.

the key-value pairs are *not* embedded in its leaf nodes; instead, it stores a pointer to the key-value pairs. This incurs additional pointer dereference, which is *not* an NVM friendly design. Consequently, PDL-ART's range scan performance is up to 2× slower than the PACTree. Although PACTree uses PDL-ART as its search layer, it uses a slotted leaf node. With the slotted leaf nodes, NVM allocation overhead in PACTree is significantly reduced, and moreover, it enables a better spatial locality that efficiently uses the hardware cache and the hardware prefetcher to hide the latency. This enables a high write and range scan performance in PACTree.

6.2 Performance on Low Bandwidth NVM machine

Figure 11 presents the performance of PACTree and the other persistent indexes on a two-socket NVM server with lower cumulative NVM bandwidth and capacity than our default evaluation platform. Each socket consists of 16 physical cores and 256 GB NVM (2×128GB). The cumulative NVM bandwidth is about 3× lesser than our default evaluation platform. We present the performance with 32 threads. The performance gap between PACTree and PDL-ART is widened by up to 0.5× for write-intensive workloads and 1.5× for read-intensive workloads. *With lower NVM bandwidth, asynchronous search layer update becomes much more critical as it reduces the write latency in the critical path and consumes relatively less NVM bandwidth in the critical path.*

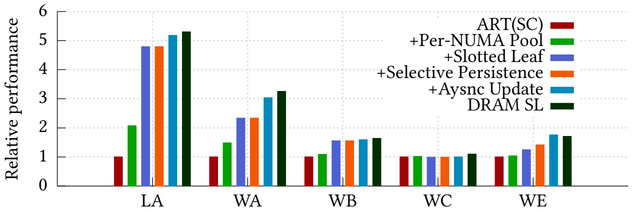

Figure 12. Factor analysis of PACTree: 64M Zipfian string keys with 28 concurrent threads.

6.3 Factor Analysis for PACTree Design

Figure 12 presents the factor analysis on PACTree. We start with the PDL-ART and add proposed design features.

+ Per-NUMA pool. For write-intensive workloads, adding per-NUMA NVM pool improves the performance by up to 2×. With per-NUMA pool, writers allocate memory locally, and they can fully utilize the NVM bandwidth of the system.

+ Slotted leaf nodes. Incorporating slotted leaf nodes to PDL-ART improves the performance by up to 2.5× across all workloads except for read-only Workload C. Unlike the leaf nodes in PDL-ART, slotted leaf nodes do not incur NVM memory allocation for every insert operation, thereby amortizing the allocation overhead. For Workload C, adding slotted leaf nodes shows a slight dip (<10%) in performance as it requires one fingerprint match operation, indirection to key-value pair, and full key comparisons.

+ Selective persistence. Selective persistence for data node improves the scan performance by up to 11% because it does not incur blocking by issuing persistence instruction.

+ Asynchronous search layer update. Adding asynchronous update improves the performance by up to 30% for write-intensive workloads. Updating the search layer in the background significantly reduces the critical path latency of write operations.

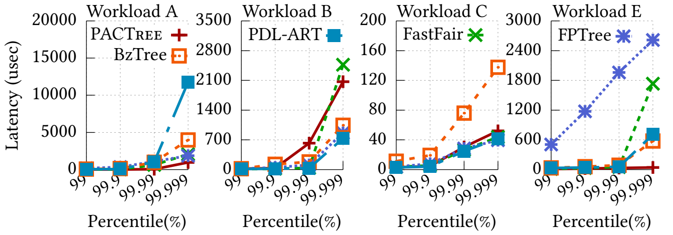

Figure 13. Tail latency comparison for integer keys with uniform distribution and 56 threads.

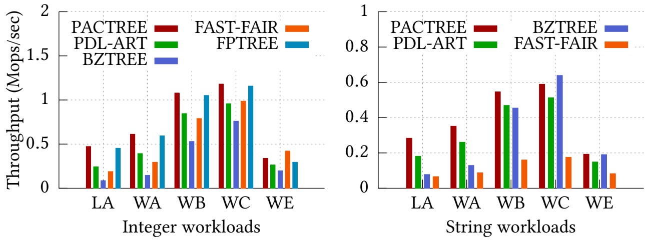

Figure 14. Single threaded performance of persistent indexes.

- Search layer on DRAM. We also evaluate the effect of DRAM for the search layer as in the previous works [1, 8, 46, 57, 75, 76] for reference. Our evaluation shows that the DRAM-based search layer is not that beneficial for PACTree (less than 10% increase). As the search layer is updated asynchronously in the background, there is no performance benefit in placing it on the DRAM. The saved DRAM consumption can be used for application or for further performance optimization at the index level (*e.g.*, caching hot items [64]).

6.4 Tail Latency

We evaluate the latency distributions of the persistent indexes. We sampled 10% of operations to reduce the measurement overhead as in the previous study [44]. As shown in Figure 13, PACTree shows low tail latency up to 20× in write-intensive workloads and shows comparable latency in the read-intensive workloads due to its asynchronous update, amortized NVM allocation overhead by using B+tree leaf nodes, and trie-based search layer. BzTree and PDL-ART show high tail latency because of their high memory allocation overhead, and BZTree also suffers from indirection overhead because of its lock-free design. FPTree shows the worst tail latency for Workload E because it needs additional sorting and filtering for scan operations.

6.5 Single Thread Performance

We compare the single-threaded performance of the persistent indexes. As shown in Figure 14, PACTree shows similar or up to 3× better performance than the other persistent indexes because PACTree's optimistic version locks do not impose any overhead when there is no contention.

6.6 Skew Test

Figure 15 shows the performance of PACTree on the Zipfian workloads with 28 threads and 56 threads. We evaluate PACTree using two write-intensive workloads: (1) 50% lookup + 50% update and (2) 50% lookup + 50% insert with varying the Zipfian coefficient values. In the 50% lookup + 50% update workload, PACTree shows better performance for highly skewed workloads because of the better cache-locality of the data nodes. Also, optimizations in PACTree make the critical path in update shorter and thus delivering a good performance. In the 50% lookup + 50% insert workload, PACTree shows stable performance as it can reduce the critical path by updating the search layer asynchronously.

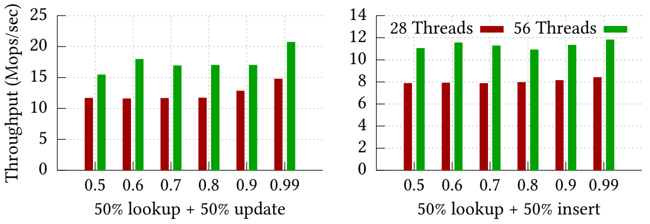

Figure 15. Performance of PACTree for varying Zipfian coefficient.

6.7 Impact of Asynchronous Search Layer Update

One potential downside of asynchronous search layer update is that a long traversal from a jump node to a target node when two layers are not synchronized. To investigate this, we ran write-intensive Workload A with 112 threads and measured the distance between a jump node and a target node. Our results show that in 68% of cases, the search layer directly leads to the target layer. In 30% of cases, it requires only one hop traversal from a jump node to a target node.

6.8 Recovery

We conducted the recovery evaluation on PACTree by injecting a crash 100 times using SIGKILL, similar to the previous studies [11, 39, 44]. We confirmed that our PACTree successfully recovered from every crash and ensured that all previously written keys could be accessed.

7 Conclusion

Designing a high performance persistent index goes beyond just making DRAM index crash consistent. There are NVM unique design factors that are critical in NVM rather than in DRAM. Based on our thorough analysis of prior persistent indexes on real NVM hardware, we proposed the PAC guidelines. Following the PAC guidelines, we designed PACTree, which packs partial keys in internal nodes, and updates internal nodes in an asynchronous and concurrent manner. Our evaluation shows that PACTree significantly outperforms state-of-the-art persistent range indexes in terms of performance, scalability, and tail latency.

Acknowledgement

We thank our shepherd Kang Chen and the anonymous reviewers for their insightful comments. This work was supported in part by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035) and National Research Foundation of Korea (NRF) (No. NRF-2021R1A6A3A03046359).

References

[1] Key/Value Datastore for Persistent Memory. https://github.com/pmem/pmemkv.

[2] perf: Linux profiling with performance counters , 2020. https://perf.wiki.kernel.org/index.php/Main_Page.

[3] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. November.

[4] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A High-performance Latch-free Range Index for Non-volatile Memory. In *Proceedings of the 44th International Conference on Very Large Data Bases (VLDB)*, Rio De Janerio, Brazil, August 2018.

[5] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind Logging. In *Proceedings of the 42nd International Conference on Very Large Data Bases (VLDB)*, New Delhi, India, March 2016.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 753–766, Renton, WA, July 2019.

[7] Shimin Chen and Qin Jin. Persistent B+-trees in Non-volatile Main Memory. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, Hawaii, USA, September 2015.

[8] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. Utree: A persistent b+-tree with low tail latency. *Proc. VLDB Endow.*, 13(12):2634–2648, July 2020.

[9] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free Concurrent Level Hashing for Persistent Memory. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 799–812, Boston, MA, July 2020.

[10] Dave Chinner. xfs: updates for 4.2-rc1., 2015.

[11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, March 2011.

[12] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-grain checkpointing with in-cache-line logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 441–454, New York, NY, USA, April 2019. Association for Computing Machinery.

[13] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, Indiana, USA, June 2010. ACM.

[15] Mohammad Dashti, Alexandra Fedorova, Justin R. Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 381–394, Houston, TX, March 2013.

[16] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. POSEIDON: Safe, Fast and Scalable Persistent Memory Allocator. In *Proceedings of the 21st ACM/IFIP International Middleware Conference (Middleware 2020)*, 2020.

[17] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[18] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, April 2014.

[19] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004. No. UCAM-CL-TR-579.

[20] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, pages 1304–1318, Tokyo, Japan, August 2020.

[21] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.*, 14:626–639, 2021.

[22] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[23] Takahiro Hirofuchi and Ryousei Takano. The preliminary evaluation of a hypervisor-based virtualization mechanism for intel optane dc persistent memory module. *arXiv preprint arXiv:1907.12014*, 2019.

[24] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 187–200, Oakland, California, USA, February 2018.

[25] Intel. Chapter 12. Intel Optane DC Persistent Memory, Intel 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures_optimization-reference-manual.html.

[26] INTEL. poolset - persistent memory pool configuration file format. https://pmem.io/pmdk/manpages/linux/v1.4/poolset/poolset.5.

[27] INTEL. Persistent Memory Development Kit, 2019. http://pmem.io/.

[28] INTEL. PMDK man page: pmemobj_alloc, 2019. http://pmem.io/pmdk/manpages/linux/v1.5/libpmemobj/pmemobj_alloc.3.

[29] Intel. PMWatch (PersistentMemoryWatch), 2020. https://github.com/intel/intel-pmwatch.

[30] Joseph Izraelevitz, Hammurabi Mendes, and Michael Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Proceedings of the 30st International Conference on Distributed Computing (DISC)*, Paris, France, September 2016.

[31] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[32] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[33] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*, Virtual, October 2020.

[34] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Non-volatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[35] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[36] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[37] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[38] 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 *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February–March 2017.

[39] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[40] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 38–49, Brisbane, Australia, April 2013.

[41] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of Practical Synchronization. In *Proceedings of the International Workshop on Data Management on New Hardware*, pages 3:1–3:8, San Francisco, California, June 2016.

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

[43] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, pages 574—-587, Los Angeles, CA, August 2019.

[44] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating Persistent Memory Range Indexes. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, Los Angeles, CA, August 2019.

[45] Jihang Liu, Shimin Chen, and Lujun Wang. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.*, 13(7):1078–1090, 2020.

[46] Jihang Liu, Shimin Chen, and Lujun Wang. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

[47] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 133–148, Santa Clara, California, USA, February 2016.

[48] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query optimized persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, Virtual, February 2021.

[49] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*, pages 183–196, Bern, Switzerland, April 2012.

[50] Ajit Mathew and Changwoo Min. HydraList: A Scalable In-Memory Index Using Asynchronous Updates and Partial Replication. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

[51] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, 2007.

[52] Paul E. McKenney, Jonathan Appavoo, Andy Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-Copy Update. In *Ottawa Linux Symposium*, OLS, 2002.

[53] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, Boston, MA, February 2019.

[54] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. SQLite Optimization with Phase Change Memory for Mobile Applications. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, pages 1454–1465, Hawaii, USA, September 2015.

[55] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. EuROSYS16.

[56] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory Management Techniques for Large-scale Persistent-main-memory Systems. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, TU Munich, Germany, August 2017.

[57] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*, San Francisco, CA, USA, June 2016.

[58] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. SQL Statement Logging for Making SQLite Truly Lite. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, pages 513–525, TU Munich, Germany, August 2017.

[59] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 497–514, Shanghai, China, October 2017.

[60] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[61] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[62] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory i/o primitives. In *Proceedings of the International Workshop on Data Management on New Hardware*, pages 1—-7, Amsterdam, The Netherlands, July 2019.

[63] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, California, USA, February 2011.

[64] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A black-box approach to numa-aware persistent memory indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, November 2021.

[65] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 461–472, Paris, France, April 2018.

[66] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*, pages 473–488, Houston, TX, USA, June 2018.

[67] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508, Virtual, October 2020.

[68] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC'19)*, pages 1—-19, Denver, CO, November 2019.

[69] Matthew Wilcox. Add Support for NV-DIMMs to Ext4., 2014.

[70] X. Wu and A. L. N. Reddy. Scmfs: A file system for storage class memory. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC'14)*, Seattle, WA, November 2011.

[71] Xingbo Wu, Fan Ni, and Song Jiang. Wormhole: A Fast Ordered Index for In-memory Data Management. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, pages 18:1–18:16, Dresden, Germany, March 2019.

[72] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[73] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February 2016.

[74] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2020.

[75] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February 2015.

[76] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. DP-Tree: Differential Indexing for Persistent Memory. *Proc. VLDB Endow.*, 13(4):421–434, December 2019.

[77] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, October 2018.