# Evaluating Persistent Memory Range Indexes

Lucas Lersch
TU Dresden & SAP SE
lucas.lersch@sap.com

Xiangpeng Hao
Simon Fraser University
xha62@sfu.ca

Ismail Oukid*
Snowflake Computing
ismail.oukid@snowflake.com

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

Thomas Willhalm
Intel Deutschland GmbH
thomas.willhalm@intel.com

## ABSTRACT

Persistent memory (PM) is fundamentally changing the way database index structures are built by enabling persistence, high performance, and (near) instant recovery all on the memory bus. Prior work has proposed many techniques to tailor index structure designs for PM, but they were mostly based on volatile DRAM with simulation due to the lack of real PM hardware. Until today is it unclear how these techniques will actually perform on real PM hardware.

With the recent released Intel Optane DC Persistent Memory, for the first time, this paper provides a comprehensive evaluation of recent persistent index structures. We focus on $B^+$-Tree-based range indexes and carefully choose four representative index structures for evaluation: wBTree, NV-Tree, BzTree and FPTree. These four tree structures cover a wide, representative range of techniques that are essential building blocks of PM-based index structures. For fair comparison, we used an unified programming model for all trees and developed *PiBench*, a benchmarking framework which targets PM-based indexes. Through empirical evaluation using representative workloads, we identify key, effective techniques, insights and caveats to guide the making of future PM-based index structures.

**PVLDB Reference Format:**
Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, Thomas Willhalm. Evaluating Persistent Memory Range Indexes. *PVLDB*, 13(4): 574-587, 2019.
DOI: https://doi.org/10.14778/3372716.3372728

## 1. INTRODUCTION

Next-generation, scalable persistent memory (PM) promises low latency (comparable to DRAM's), byte-addressability, scalability (large capacity) and non-volatility on the memory bus. These properties make PM attractive for index structures (e.g., $B^+$-Tree and its variants) in OLTP systems: the index can be directly accessed and persisted in PM and be recovered (nearly) instantly, saving much rebuild/loading time, improving performance (compared to a disk-based index), and easing the effort to manage a large index.

PM exhibits several properties that are distinct from DRAM and flash memory. It has a higher endurance than flash, but not unlimited like DRAM. Its latency is comparable and higher than DRAM, but still an order of magnitude lower than flash. It also presents bandwidth lower than DRAM, and asymmetric read/write speeds [23]. Blindly moving existing index structures to run on PM would not reap PM's real benefits for these structures. This necessitates nontrivial efforts in redesigning index structures for PM. There have been numerous proposals [6,9,10,18,25,32,38,42,47] that tailor index structures for PM. However, prior work mostly had to base on volatile DRAM and emulation due to the lack of real PM hardware when they were developed. Thus, it is unclear how well the proposed approaches will in fact work on real PM hardware.

In this paper, we provide a comprehensive evaluation of range indexes on real PM hardware based on the recently released Intel Optane DC Persistent Memory Modules (DCPMM). DCPMM uses the 3D XPoint technology [12] which scales to large capacity (up to 512GB per DIMM) and provides latency in the range of that of DRAM. It is so far the only scalable PM product that is in mass production and we expect it to be mainstream in the near future.

The goals of this work are to (1) qualitatively and quantitatively compare range indexes designed specifically for PM, (2) understand the behavior and impact of different design decisions in the context of real PM hardware, and more importantly, (3) distill useful insights and design guidelines for tree structures in PM. To achieve these goals, we have developed **PiBench**[1], a ***p***ersistent ***i***ndex benchmarking framework. PiBench defines a set of common interfaces (`lookup`, `insert`, `update`, `delete`, `scan`) supported by index structures and implements unified, highly customizable benchmarks for all data structures under evaluation. Using a unified framework allows us to fairly compare multiple index structures and rule out the impact of different benchmark implementations. Through a shared library, PiBench can support any index structures—including hash tables, tries and trees—that support the common operations.

We focus on $B^+$-Tree [7] based index structures that support range scans, because they are arguably the most widely used index in OLTP systems, have received the most attention, and have the most mature techniques among all persistent index types [6,9,10,18,32,42,47]. We identify and evaluate using PiBench four recent and representative proposals, including BzTree [6], FPTree [32], NV-Tree [47] and wBTree [10]. Although we are evaluating only four tree structures, they cover a wide range of techniques in various dimensions. For example, in terms of concurrency control, our selection covers both lock-based (FPTree and NV-Tree), lock-free (BzTree), and hardware transactional memory (HTM) based (FPTree) approaches. We describe and compare these index structures and their key techniques in Section 3.

Our evaluation results obtained using representative workloads (cf. Section 5) revealed several important insights, highlighted below:

* Work done while employed by SAP SE.

[1] Available at `https://github.com/wangtzh/pibench`.

*Proceedings of the VLDB Endowment,* Vol. 13, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3372716.3372728

- Contrary to the estimates found in prior work [40, 49], our results corroborate with recent work [37] that PM bandwidth is a scarce resource and has significant impact on performance. Data structures thus need to be designed to not exhaust the available PM bandwidth. This is especially true for machines that are not fully populated, i.e., with only a few PM DIMMs.
- Designing indexes (and data structures in general) for PM requires using a sound programming model provided by some PM library [19, 31, 42] for correctness and usability. This entails non-trivial overhead but is largely sidestepped by prior work. The result is a significant slowdown on real PM compared to the originally reported numbers using emulations on DRAM. The interactions between data structures and PM libraries must be carefully coordinated.
- Despite the different designs, there are several effective key building blocks and principles that should be followed when designing indexing structures for PM; they are largely orthogonal and can be applied individually depending on the need.

In the main sections, we elaborate and present more detailed findings and insights. To the best of our knowledge, this is the first and most comprehensive evaluation of OLTP index structures on real, next-generation PM. PiBench is the first and only framework targeted at evaluating index structures with support for collecting metrics on real PM.

Next, we first give background on PM in Section 2, including its hardware characteristics and implications on software. We then introduce and compare the indexes under evaluation and our benchmark framework in Sections 3 and 4, respectively. Section 5 presents our empirical evaluation results and analysis. We discuss our findings and insights obtained from this evaluation in Section 6. Section 7 discusses related work, and Section 8 concludes this paper.

## 2. PERSISTENT MEMORY

There are several types of PM based on different materials, such as memristor [35], STT-RAM [17], phase change memory (PCM) [43], 3D XPoint [12] and DRAM/flash-based NVDIMM [5, 39]. Despite the different underlying materials, the common features offered by PM include (1) byte-addressability, (2) non-volatility and (3) performance in the range of DRAM's. They can be placed on the memory bus, thus appear to software as normal memory and can be accessed directly using `load` and `store` instructions.

Optane DC PM and NVDIMM are the only commercially available PM products, and we expect next-generation, scalable PM (e.g., Optane DC) to be mainstream soon. Therefore, we target Optane DC PM based on the 3D XPoint technology [12]. The rest of this section provides the necessary background on Optane DC PM's properties and discusses PM programming models.

### 2.1 Optane DC Persistent Memory

**Performance.** Optane DC PM scales much better than DRAM, thus it is capable of providing much larger capacity (up to 512 GB per DCPMM); a single CPU can be equipped with 3 TB of DCPMM. However, its read/write latency is higher than DRAM's and it exhibits asymmetric read/write latency (writes being slower). Compared to DDR4 DRAM, Optane DC PM has a 300 ns read latency, $\sim$4$\times$ higher that that of DRAM (75 ns). Optane DC PM exhibits peak sequential read and write bandwidth of 40 GB/s and 10 GB/s, respectively. These are respectively $\sim$3$\times$ and $\sim$11$\times$ times lower than those of DDR4 DRAM. This gap widens even more for random read and write bandwidth to respectively $\sim$8$\times$ (7.4 GB/s) and $\sim$14$\times$ (5.3 GB/s) lower bandwidth than DDR4 DRAM.

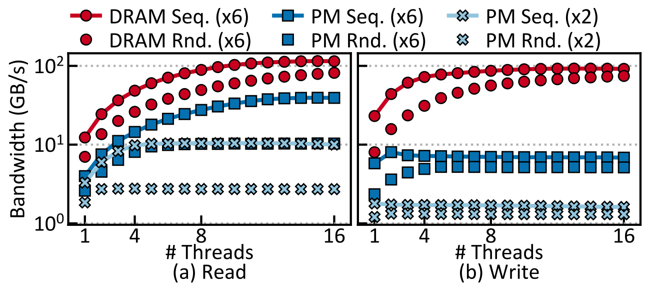

**Figure 1:** DRAM and Optane DCPMM bandwidth when a server is fully ($\times$6 modules) and partially ($\times$2 modules) populated.

Given the performance gap between DRAM and PM, it becomes more important to leverage more memory channels (i.e., equip more DCPMMs) to reach the peak bandwidth. Figure 1 compares the bandwidth under two and six DCPMMs with a varying number of threads. The CPU supports six channels, each of which has two slots, one for DRAM and one for PM. Therefore, the number of DCPMMs also indicates the number of memory channels. Similar to DRAM, adding more DCPMMs significantly increases read performance for PM, while the impact for write bandwidth is relatively smaller. It is worth noting that in most cases using two threads is enough to saturate PM write bandwidth. In our experiments, unless otherwise noted, we use six DCPMMs. As we will see in Section 5, PM bandwidth is a scarce resource, and higher bandwidth (more DCPMMs) is critical in obtaining high index performance. Nevertheless, having as many DCPMMs as possible is not always better, as the optimal setup highly depends on the use case. As an example, one might benefit from less DCPMMs in favor of more DRAM modules since the amount of available memory slots is limited per CPU. In such case, this would imply trading the lower costs and increase in bandwidth of PM for a more expensive setup and larger DRAM capacity, benefiting from its lower latency access.

It is difficult to accurately measure write latency for DCPMM, as writes "succeed" once they reach the memory controller buffers from CPU caches [33]. Measuring the latency of cache-flushing instructions does not give real latency to the actual PM device. A comprehensive evaluation on DCPMM raw performance can be found elsewhere [23]; we focus on understanding how more complex, range indexes perform on DCPMM.

**Operating modes.** Optane DCPMM can be configured to run in two modes: *Memory* and *App Direct* [20]. Both allow direct access to PM by normal `load` and `store` instructions and can leverage CPU caches for higher performance. Under the Memory mode, DCPMM acts as large memory *without* persistence; DRAM is used as a cache to hide the longer latency. The App Direct mode provides persistence. There is no DRAM cache in front to hide the high latency; the application should judiciously use PM and handle persistence, recovery, concurrency and optimize for performance. More details can be found in Intel manuals [3].

The crux of persistent indexes is leveraging non-volatility and guaranteeing failure atomicity, so the Memory mode is not useful for them, as data will be wiped across reboots. Therefore we configure DCPMM to run in the App Direct mode. The system can also feature a certain amount of volatile DRAM, and it is up to the software (indexes in our case) to determine the roles of DRAM. As Section 3 describes, some indexes are entirely in PM, while some leverage both DRAM and PM. Note, however, that this is not to be confused with the aforementioned "Memory" mode where DRAM is used as a cache for PM and is transparent to persistent data structures.

## 2.2 Programming Persistent Memory

The use of PM introduces new challenges in data persistence, memory management and concurrency. Solving these challenges requires the use of a sound programming model [32] that consists of the use of cacheline flush instructions and PM-aware pointers, allocators and concurrency control mechanisms. This constitutes a key part in designing persistent data structures and is usually done using PM programming libraries [19, 31, 42]. This section gives an overview of these issues; we elaborate in detail in Sections 3 and 5.

**Persistence.** Since CPU caches are volatile and there is no way in software to prevent cachelines from being evicted, data must be properly flushed from the cache to PM eagerly for safe persistence. This can be done using the `CLFLUSH`, `CLFLUSHOPT` or `CLWB` instructions, which will flush the specified cacheline contents to the memory controller (write buffers) that cover PM. Through asynchronous DRAM refresh [33] the write buffers are guaranteed to be persisted in PM upon power failure. `CLFLUSHOPT` and `CLFLUSH` will evict the cachelines being flushed, so they can significantly impact performance; `CLWB` is a new instruction for PM that flushes a cacheline without evicting it. Also, applications that rely on a specific ordering of writes to guarantee consistency must issue `SFENCE` to avoid stores from being re-ordered by the CPU. Moreover, modern CPUs only guarantee 8-byte atomic writes[2] in a single cycle. As a result, data chunks larger than 8 bytes might be written back to PM in separate cycles, leading to partial writes upon power failure.

**Memory management.** PM can be mapped to the application's address space using a PM-aware file system [2,4,14,45] that provides direct access without file system caching, using the `mmap` interface. The application then uses virtual memory pointers to access data in PM. However, `mmap` does not guarantee the application will obtain the same address space across reboots, invalidating all the stored pointer values. So the system needs to be able to correctly store and transform pointers to use the new address space upon recovery. This is typically handled by some PM programming library, e.g., by recording only offsets in PM and generating pointers on-the-fly by adding the offset to a base address. We use the Ext4 file system with Direct Access (DAX) [1] to manage PM.

PM applications also require the memory allocator to properly handle transfer of memory ownership to prevent *permanent* PM leaks. The allocator must guarantee that an allocated PM block is atomically "given" to the application and never leaves it in a state where the memory is tracked by neither the application nor the allocator. Most persistent allocators [8, 19, 31] follow an *allocate-activate* approach using a `posix_memalign`-like interface. A protocol needs to be in place for the allocator to determine the right memory ownership upon recovery.

**Concurrency and recovery.** After a reboot, DRAM-only structures can be re-created or recovered from storage without inconsistencies, and start with a clean slate. But PM applications need to recover both data and program states (e.g., critical sections) back to a consistent state upon reboot, because they are all persistent in PM. For instance, in a B$^+$-Tree that uses lock-based concurrency control, a split operation that manipulates multiple pointers may be done in a critical section. Since the lock is typically also PM-resident, a crash may cause the tree to hold the lock forever and be in an inconsistent state. One solution is to devise a recovery mechanism that releases the lock and rolls back the changes upon recovery. Similarly, a lock-free B$^+$-Tree may expose intermediate states (e.g., half-finished split [27]). As a result, the synchronization mechanism needs to be tailored for PM to ensure correctness.

[2] Not to be confused with atomic visibility, which can be achieved at larger sizes with instructions such as `CMPXCHG16B`.

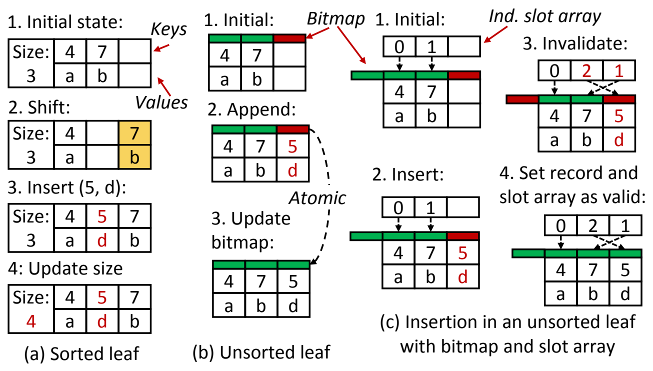

**Figure 2:** Comparison of inserting a record `d` with key `5` in (a) sorted node, (b) unsorted node and (c) unsorted node with indirection slot array. Using unsorted node reduces writes but requires linear search for lookup, which can be avoided by using an indirection slot array.

## 3. PERSISTENT B+-TREE STRUCTURES

In this section, we survey representative PM-based B$^+$-Tree [7] indexes, including wBTree [10], NV-Tree [47], BzTree [6] and FP-Tree [32]. For completeness we discuss other indexes in Section 7.

### 3.1 Write-Atomic B$^+$-Tree (wBTree)

wBTree [10] is a persistent, single-threaded B$^+$-Tree that achieves high performance by reducing cacheline flushes and writes to PM. Traditional B$^+$-Tree nodes are sorted for faster binary search. However, as Figure 2(a) shows, keeping a node sorted requires a shift of data to make place for the new key, which might leave the node in an inconsistent state upon crashes, and incurs more (expensive) PM writes. wBTree solves this problem with unsorted nodes proposed in prior work [9]. Figure 2(b) illustrates the idea. A bitmap is used to indicate if each slot contains valid (green box in the figure) record or not (red box). The new record is inserted into a free slot (out-of-place), and the bitmap is atomically modified using 8-byte writes to set the validity of the inserted record. Using unsorted nodes reduces the number of (expensive) PM writes and eases implementation, but requires linear search for lookups, which might be more expensive than a binary search. Nevertheless, as we will see later, the use of unsorted nodes is a common and effective design in PM trees.

To enable binary search (thus reducing PM accesses), wBTree uses an indirection slot array in each node, as shown in Figure 2(c). Each entry of the array records the index position of the corresponding key in sorted order, i.e., the n-th array element will "point" to the n-th smallest key by recording the key's index into the key-value slots. In the example, after inserting key `5`, in step 3 the bitmap needs to be modified so that the third element records the position of key `7`, which is stored as the second element (index 1) in the key-value storage area. One bit (left-most box in the figure) in the bitmap is reserved to indicate the validity of the array. wBTree relies on the atomic update of the bitmap to achieve consistency, and on logging for more complex operations such as node splits. After inserting the record out-of-place in a free slot, the indirection slot array is flagged as invalid and updated, as shown in step 3 of Figure 2(c). In case of a failure, the indirection slot array will be detected as invalid and reconstructed upon recovery. Finally, the bitmap is atomically updated to set both the indirection slot array and the new record as valid. This last step imposes that the bitmap be no larger than 8 bytes. When the indirection slot array is smaller than 8 bytes, the bitmap could be removed as the indirection slot array can be atomically updated and serve as the validity flag.

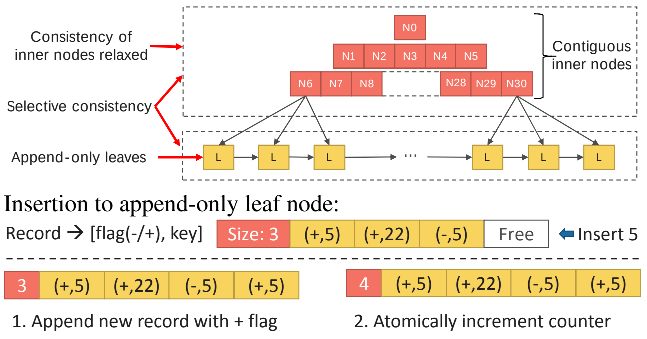
Consistency of inner nodes relaxed
Selective consistency
Append-only leaves
Contiguous inner nodes
N0
N1 N2 N3 N4 N5
N6 N7 N8 N28 N29 N30
L L L ... L L L
Insertion to append-only leaf node:
Record → [flag(-/+), key]
Size: 3 (+,5) (+,22) (-,5) Free
Insert 5
3 (+,5) (+,22) (-,5) (+,5)
4 (+,5) (+,22) (-,5) (+,5)
1. Append new record with + flag
2. Atomically increment counter

**Figure 3:** NV-Tree architecture (top) and insertion process (bottom).

## 3.2 NV-Tree

NV-Tree [47] proposes the concept of *selective consistency*, which as shown in Figure 3, enforces the consistency of leaf nodes and relaxes that of inner nodes. This design simplifies implementation and reduces consistency costs by avoiding many cacheline flushes. Inner nodes, however, have to be rebuilt upon recovery because the copy in PM might be inconsistent and unable to guide lookups correctly. We note that inner nodes could also be placed in DRAM since their consistency is not enforced. Similar to the wBTree, NV-Tree also uses unsorted leaf nodes with an append-only strategy to achieve fail-atomicity. Figure 3(bottom) shows an example of an insertion in an NV-Tree leaf node. The record is directly appended with a positive flag (or a negative flag in case of a deletion) regardless of whether the key exists or not. Then, the leaf counter is atomically incremented to reflect the insertion. To lookup a key, the leaf node is scanned backwards to find the latest version of the key: if its flag is positive, then the key exists and is visible; otherwise, the key has been deleted. The inner nodes are stored contiguously to abstract away pointers and improve cache efficiency. However, this implies the need for costly rebuilds when a parent-to-leaf node needs to be split. To avoid frequent rebuilds, inner nodes are rebuilt in a sparse way, which may lead to high memory footprint. As inner nodes are immutable (except parent-to-leaf nodes) once they are built, threads can access them without locking and only need to take locks at the leaf and their parents level when traversing the tree.

## 3.3 BzTree

BzTree [6] is a lock-free B$^+$-Tree for PM that uses persistent multi-word compare-and-swap (PMwCAS) [42] to handle concurrency and ease implementation. PMwCAS is a general-purpose primitive that allows atomically changing multiple arbitrary 8-byte PM words in a lock-free manner with crash consistency. To achieve this, PMwCAS uses a two-phase approach. In Phase 1, it uses a descriptor $d$ to collect the "expected" and "new" values for each target word, persist the descriptor, and atomically installs (using single-word `CAS`) a pointer to the descriptor on each word. If Phase 1 succeeded, Phase 2 will install the new values; otherwise the operation is aborted with all changes rolled back.

BzTree uses PMwCAS for insert, delete, search, scan, and structural modification operations which may need to change multiple PM words. Because of the use of PMwCAS, while being lock-free, BzTree implementation is easier to understand than typical lock-free code. PMwCAS ensures that any multi-word changes are done atomically and recovery is transparent to BzTree, removing the need for customized logic for logging and recovery.

As Figure 4 shows, BzTree stores both inner and leaf nodes in PM. Inner nodes are immutable (copy-on-write) except for updates

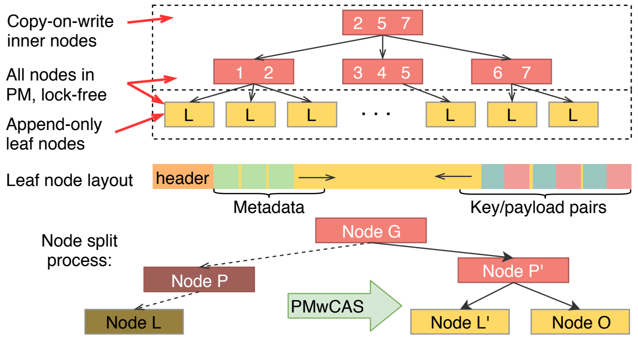

**Figure 4:** Overview of the architecture of the BzTree.

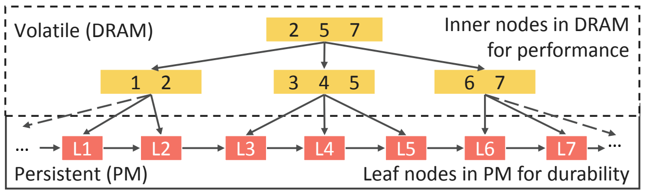
Volatile (DRAM)
Inner nodes in DRAM for performance
2 5 7
1 2
3 4 5
6 7
... L1 L2 L3 L4 L5 L6 L7 ...
Persistent (PM)
Leaf nodes in PM for durability

**Figure 5:** FPTree's hybrid PM-DRAM design.

to existing child pointers; leaf nodes can accommodate inserts and updates. Inserting to a parent node causes it to be replaced with a new one that contains the new key. Then, an update in the grandparent node is conducted to point to the new parent node. Splits can propagate up to the root and grow the tree. Records in inner nodes are always sorted, while records in leaf nodes are not. Initially, records are inserted to the free space serially. Periodically leaf nodes get consolidated (sorted) and subsequent inserts may continue to insert into the free space serially. After searching the sorted area (using binary search), the tree must linearly search the unsorted area to get correct result. The design rationale is that inner nodes are not updated as often as leaf nodes and should be search-optimized; leaf nodes, however, need to be write-optimized.

## 3.4 Fingerprinting Persistent Tree (FPTree)

Unlike the other trees being evaluated, FPTree [32] uses both DRAM and PM to achieve near-DRAM performance. As Figure 5 shows, it stores inner nodes in DRAM, and leaf nodes in PM. This way, FPTree accelerates lookup performance while maintaining persistence of primary data (leaf nodes), as only leaf accesses are more expensive during a tree traversal compared to a fully transient counterpart. The rationale behind is that while losing leaf nodes leads to an irreversible loss of data, inner nodes can always be rebuilt from leaf nodes. Since the inner nodes must be rebuilt upon recovery, FPTree trades recovery time for higher runtime performance.

FPTree uses fingerprints to accelerate search. They are one-byte hashes of in-leaf keys, placed contiguously in the first cacheline-sized piece of the leaf node. FPTree also uses unsorted leaf nodes with in-leaf bitmaps [9], such that a search iterates linearly over all valid keys in a leaf. A search will scan the fingerprints first, limiting the number of in-leaf key probe to one on average, which significantly improves performance. FPTree applies different concurrency control methods for the tree's transient and persistent parts. It uses hardware transactional memory (HTM) and fine-grained locks for inner and leaf nodes, respectively. Such *selective concurrency* design solves the apparent incompatibility of HTM and persistence primitives required by PM such as cacheline flushing instructions which always cause HTM transactions to abort directly.

**Table 1:** Comparison of key design choices of the tree structures being evaluated.

| | Architecture | Node structure | Concurrency |
|---|---|---|---|
| **wBTree** | PM-only | Unsorted | Single-threaded |
| **NV-Tree** | PM-only (optionally hybrid PM-DRAM) | Unsorted leaf nodes; inconsistent inner nodes | Locking |
| **BzTree** | PM-only | Partially unsorted leaf; sorted inner nodes | Lock-free (PMwCAS [42]) |
| **FPTree** | DRAM (inner nodes) + PM (leaf nodes) | Unsorted leaf nodes | Selective (HTM + locking) |

## 3.5 Discussion

We summarize the design trade-offs in Table 1. The design space includes (1) deciding the roles of PM and DRAM, (2) achieving safe persistence while reducing consistency cost and PM accesses, and (3) handling concurrency. The techniques are mostly orthogonal and can be used as building blocks to design novel PM data structures. Here we compare the four trees in terms of each design decision.

**Architecture.** FPTree and NV-Tree can leverage both DRAM and PM to store inner and leaf nodes in DRAM and PM, respectively. This removes the need to access PM until the end of the traversal at the leaf level. This approach can achieve near DRAM performance, but trades off recovery time as inner nodes must be rebuilt upon recovery. The other two indexes, wBTree and BzTree, store the entire tree in PM, thus may suffer longer lookup time.

**Node structure.** All the evaluated trees use unsorted nodes to reduce consistency costs and accesses to PM, at the expense of potentially more expensive lookup. FPTree solves this problem with bitmaps and fingerprints; NV-Tree and BzTree have to scan unsorted nodes linearly. NV-Tree allows inner nodes to be inconsistent to reduce cacheline flushes. BzTree periodically consolidates leaf nodes in sorted order; inner nodes are always kept sorted using copy-on-write (CoW) to accelerate traversal using binary search.

**Concurrency.** Except wBTree which is single-threaded, the other trees employ different approaches. NV-Tree uses locking. FPTree uses locking for leaf nodes and HTM for inner nodes (selective concurrency). BzTree is lock-free using PMwCAS. It is important to note that HTM is not compatible with PM as a cacheline flush will directly abort the transaction, as shown by the design of FPTree.

## 4. EVALUATION FRAMEWORK

We designed **PiBench** to allow unified and fair comparison of different indexes, and easy adoption by future work. As Figure 6 shows, the index being tested must be compiled into a shared library and linked to PiBench following a defined API, or through a wrapper that translates requests from PiBench's API. The API consists of a pure abstract class that encapsulates common operations (insert, lookup, delete, scan, update) and a `create_index` function for instantiating the benchmarked data structure. To use PiBench, the user only needs to derive a class that implements the API. PiBench then issues requests against the instantiated index object.

PiBench executes a `load` phase and a `run` phase, like YCSB [11]. It provides various options for customization, such as key/value sizes, the number of records to be loaded, the numbers and types of operations to be executed, and ratio of each type of operation. Keys and values are generated randomly following a chosen distribution and seed to allow reproducible executions. PiBench supports three random distributions as defined by Gray et al. [16]: `uniform`, `self similar`, and `zipfian`. Since the random distributions generate integers in a contiguous range, with the skewed distributions favoring smaller values, we apply a multiplicative hashing function [24] to each generated integer to scatter the keys across the complete integer domain, thus avoiding frequently accessed keys to be clustered together. A prefix can be prepended to keys to analyze the impact of key compression.PiBench uses multiple threads to issue requests and relies on the index under evaluation to handle concurrent accesses.

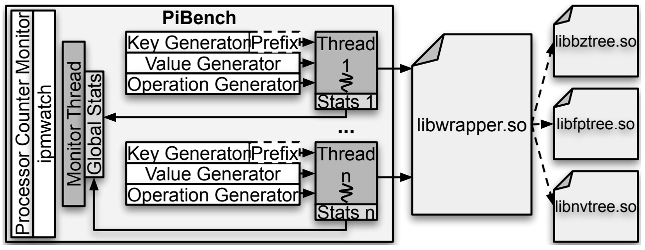

**Figure 6:** Overview of PiBench architecture.

PiBench dedicates a thread to periodically collect statistics. This allows a better understanding of performance over time by enabling standard deviation to be easily calculated in addition to the average throughput. Finally, we use the Processor Counter Monitor (PCM) library [21] and `ipmwatch`[3] to collect hardware counter metrics (such as memory accesses and cache misses). PCM measures memory traffic between CPU caches and both DRAM and DCPMMs at 64-byte granularity. The DCPMMs rely on a buffer layer to hold hot data [3, 46]. We use `ipmwatch` to measure the traffic between the buffer and the media itself, which happens at 256-byte granularity.

## 5. EXPERIMENTAL EVALUATION

This section presents our evaluation results. We first introduce the experimental setup and our implementation of the index structures. Then we present and discuss the results in detail.

## 5.1 Environment and Setup

**Hardware.** We run experiments on a Linux (5.3) server equipped with an Intel Xeon Platinum 8260L CPU, 1.5 TB of Optane DC PM ($6 \times 256$ GB DCPMMs) configured in the App Direct mode, and 96 GB of DRAM ($6 \times 16$ GB DIMMs). The CPU has 24 cores (48 hyperthreads), 36 MB of L3 cache, and is clocked at 2.40 GHz.

**Software.** To reduce the impact of different implementations, we implemented all indexes using the Persistent Memory Development Kit (PMDK) 1.7 [19]. PMDK provides primitives for managing PM, including a PM allocator. wBTree, FPTree and NV-Tree interact directly with PMDK; BzTree interacts only with PMwCAS [42], which is extended to use PMDK.[4] For DRAM allocations, we use `jemalloc` [15]. Threads are pinned to cores to avoid migration. PiBench collects the number of operations completed every 100 ms, which allows us to observe throughput over time.

We tested with different node sizes and fixed the sizes with the best performance for each tree. For FPTree and NV-Tree we use 128-record inner nodes and 64-record leaf nodes. For wBTree each inner node has 32 records and each leaf node has 64 records. We set BzTree's node size to 1 KB, same as the original paper's setup [6].

[3] Available as part of Intel VTune Amplifier 2019 since Update 5.
[4] Available at https://github.com/Microsoft/pmwcas.

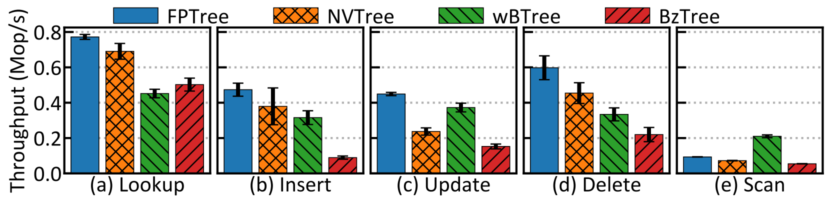

**Figure 7:** Single-thread throughput under uniform distribution. Placing inner nodes in DRAM helps much in traversal performance for FPTree and NV-Tree; wBTree performs the best in range scan as it does not require an extra sorting step.

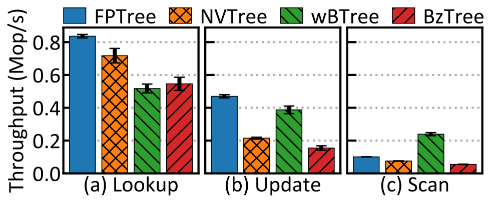

**Figure 8:** Single-thread throughput under a skewed (`self similar`) distribution with 80% accesses on 20% of data.

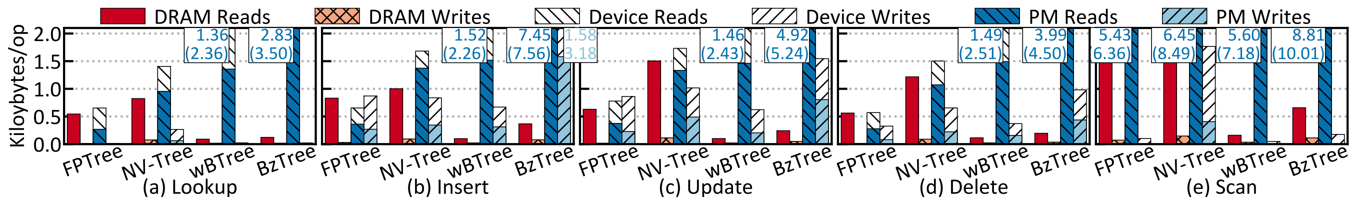

**Figure 9:** Memory accesses with a single thread under uniform distribution.

## 5.2 Index Implementations

We highlight important details for implementing the evaluated trees, especially changes we made either to make them compatible with PMDK's programming model such that they can perform on real PM, or due to necessary details not covered in original papers.[5]

**FPTree.** The original paper [32] proposed two versions: a single-thread version and a concurrent version. We focus on the concurrent version since we are most interested in multi-thread experiments. However, we note that optimizations in the single-thread version, such as allocating leaf nodes in groups, could be applied to all trees.

**wBTree.** wBTree originally uses undo-redo logs for failure atomicity [10]. We improved it with more efficient micro-logs used by FPTree [32] and implemented it using the same code template as FPTree's to reduce the impact of different implementations. We also changed wBTree to use PMDK persistent pointers.

**NV-Tree.** The original paper [47] did not cover concurrency, so we implement lock coupling. We changed NV-Tree to use PMDK persistent pointers and align records in leaf nodes to 8-byte boundaries; for 8-byte keys and values, the size of a record is 24 bytes with the validity flag. This is 7 bytes more than necessary, but gives better performance. Since the consistency of inner nodes is not enforced, we place them on DRAM to improve performance.

**BzTree.** Splits in BzTree may propagate to upper levels, replacing all the nodes along the path (CoW inner nodes). We prepare all the nodes on the split path and issue a final PMwCAS at the highest level to atomically swap in the new nodes. For this to work, we increased the size of PMwCAS descriptor size from 4 to 12 to accommodate enough memory word changes and new allocations.[6]

## 5.3 Workloads

We evaluate the indexes with individual operations (lookup, insert, update, delete, scan) and mixed workloads that combine reads and writes. All experiments are run under a uniform key distribution and a skewed (`self similar`) key distribution with a factor of 0.2 (i.e., 80% accesses focus on 20% of the keys) [16]. We then extend our experiment to cover different skew factors. Scans are performed by selecting a random initial key according to the distribution and then reading the following 100 records in ascending sorted order. We detail the mixed workload when discussing the specific tests later.

Unless otherwise specified, each run starts with a new tree pre-filled with 100 million records with 8-byte keys and 8-byte values. We then measure and report tree performance during the run phase, in which 100 million operations are executed by a specified number of threads. The numbers reported here refer only to the run phase, excluding the load phase. We use the list of operations completed in every time window (100 ms) of a single run to calculate the average throughput (depicted as the bars and points) as well as the standard deviation (depicted as the error bars) in Figures 7, 8, 11, 12, 14. We also report average and tail latency numbers collected for single-thread and multi-thread runs.

## 5.4 Single-threaded Performance

We first examine the performance of each tree under a single thread when running individual operations (i.e., 100% lookup, insert, update, delete or scan). We show throughput (in millions of operations per second) under the uniform distribution and skewed distribution with varying skew factors. We begin our discussions with each individual request type under the uniform distribution.

**Lookup.** As shown in Figures 7(a) and 8(a), trees that place inner nodes in DRAM (FPTree and NV-Tree) achieve higher throughput than trees that are fully PM based (BzTree and wBTree). FPTree's fingerprints further reduce cacheline accesses in leaf nodes to two in most cases: one for the fingerprints and bitmap, the other for the potentially matched record. This contrasts with NV-Tree which uses append-only leaves and requires scanning on average half of the leaf entries to determine if a record exists and is valid. BzTree employs a hybrid of sorted and unsorted leaf node format, so it needs to search the unsorted area linearly if the key is not found in the sorted area.

The memory access plots on Figures 9(a) and 10(a) confirm this behavior by showing more PM reads and more L3 cache misses on NV-Tree than on FPTree. For PM we differentiate between real media accesses (`Device Reads/Writes`, measured with `ipmwatch`), and accesses issued by the memory controller (`PM Reads/Writes`, measured with PCM). Note that the bars are overlaid (not stacked). In the best case, the application fully exploits the DCPMM buffer in

[5] Two authors of this paper were respectively the lead author of FPTree and PMwCAS. All trees were implemented in our best effort to get the best performance. We also improved PMwCAS for BzTree; the changes were accepted by Microsoft.

[6] These strategies were not presented in the original paper [6] but were confirmed by one of the original authors.

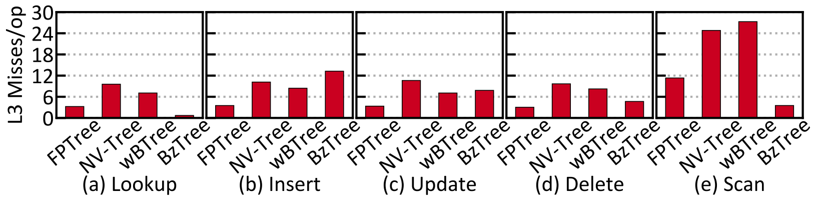

**Figure 10:** Last level cache misses with a single thread under uniform distribution.

**Table 2:** Number of cacheline flushes per operation.

| Tree/Operation | Insert | Update | Delete |
|---|---|---|---|
| FPTree | 3 | 3 | 1 |
| NV-Tree | 2 | 2 | 2 |
| wbTree | 4 | 3 | 1 |
| BzTree | 15 | 10 | 7 |

which case `Device Reads/Writes` is the same as `PM Reads/Writes`. In the worst case, `Device Reads/Writes` is four times higher, since only 64 bytes (one cacheline) is used from the 256-byte PM block that is fetched from the media into the buffer.

In Figure 10(a), BzTree incurs very few cache misses during traversal. We attribute the reason to its small node structure and search method: it uses small, 1KB pages and for 8-byte keys, our implementation uses linear search, which yields much better performance and cache behavior than binary search. wBTree performs binary search in each node using the slot arrays. Although this results in less cache misses than NV-Tree, all the cache misses pay the higher latency price of PM, resulting in lower throughput.

NV-Tree and FPTree keep inner nodes in DRAM, but NV-Tree presents more DRAM reads than FPTree (Figure 9(a)), because it keeps parent-to-leaf nodes in contiguous memory without a guaranteed fill ratio. The tree can be higher than necessary, requiring more accesses to inner nodes in DRAM. NV-Tree also incurs PM writes for reads, as it needs to acquire locks in leaf nodes stored in PM. wBTree and BzTree are purely in PM, but still present DRAM accesses. This is expected as PCM also collects DRAM accesses for managing auxiliary data while executing the operation. Finally, we note that lookup performance strongly impacts the performance of other operations as they perform a lookup prior to additional work.

**Insert.** All trees under evaluation enforce the consistency and durability of single operations using out-of-place writes (possibly within a node) and a validity bit being atomically flipped to "commit" the operation (for BzTree, this is delegated to PMwCAS). Therefore, insert, update and delete operations must always force the changes to PM using `CLWB`, making it hard for CPU caches to hide PM's high write latency. As discussed in Section 2, PM's write latency cannot be measured precisely and varies based on how far data is propagated (i.e., to the memory controller or DCPMM). This explains the lower throughput and the increased standard deviation of these operations when compared with their lookup counterparts.

Figure 7(b) shows the insert throughput. We observe insert performance is directly affected by (1) the amount of flushes per insert, (2) the needed maintenance work per insert, and (3) the overhead of node splits. Table 2 summarizes the amount of flushes needed by each operation. For all trees, each insert entails at least one flush for the record being inserted. FPTree and wBTree keep an 8-byte bitmap per node to indicate which records are valid and enable the slot of invalid records to be reused. FPTree also requires flushing the fingerprints, leading to a total of three flushes per insert. In addition to the bitmap, wBTree keeps a slotted array per node to keep the order of records and a single validity bit to indicate the validity of this slotted array. Therefore, three additional flushes are required by the wBTree (slotted array, validity bit, validity bitmap), to a total of four flushes per insert. NV-Tree requires one additional flush to update the size of the node, to a total of two flushes. BzTree uses two double-word PMwCAS operations per insert to reserve space in the leaf node and make the new insertion visible to other threads, respectively. Each PMwCAS incurs at least three flushes [42]. In total, BzTree incurs 15 flushes per insert. If the current PMwCAS conflicts with another on-going PMwCAS, it might incur more flushes as it helps finish the other operation first. We attribute BzTree's low insert performance mainly to the high number of flushes.

In BzTree, FPTree and wBTree a node split might propagate all the way up to the root level. However, for NV-Tree the inner nodes must always be completely rebuilt whenever a split happens in the parent-to-leaf level. When splitting a leaf node, two new nodes are allocated to split the records of the node that became full, causing the higher amount of PM writes seen in Figure 9(b). This operation becomes expensive in comparison to other trees, which has also an impact in the throughput standard deviation seen in Figure 7(b).

**Update.** Compared to inserts, an update only operates on an existing key. As Figure 7(c) shows, overall, the standard deviation for updates is lower than that of insert operations, due to the absence of allocations and splits. NV-Tree performs updates slower than inserts, as it handles updates as a deletion followed by an insertion. This corresponds to the higher amount of PM writes in Figure 9(c). wBTree updates are faster than inserts since each update requires one fewer flush (3 vs. 4 in Table 2), as record order in the node does not change (the key is not updated). Thus, the slotted array can be updated atomically without flushing its validity bit, as only the offset of the updated record changes, while the others remain the same. This results in lower PM writes in Figure 9. BzTree's update is faster than its insert operation, due to the absence of allocation and splits. But it still needs many flushes, leading to lower throughput.

**Delete.** As Figure 7(d) shows, delete throughputs follow a similar trend to those of lookups in Figure 7(a). The reason is that deletion for FPTree and wBTree is basically a lookup followed by flushing the validity bitmap to invalidate the record deleted. There is no deallocation or merging of nodes implemented, as data structures are more likely to grow rather than shrink. This is also the approach taken by implementations of C++ STL. In contrast to FPTree and wBTree, NV-Tree requires two flushes per deletion, one for a tombstone and one for the node size. Therefore, it has about double the amount of PM writes, as seen in Figure 9(d). For BzTree the process is similar, but it uses a PMwCAS to mark records invisible which requires multiple flushes, leading to lower performance.

**Scan.** Range scans start at a random initial key and read the following 100 records. wBTree is the only one that directly returns records in sorted order using its indirection slotted arrays. All the other trees must perform an additional sorting and filtering step to return the requested records. According to the amount of PM reads in Figure 9(e), reading less from PM (e.g., FPTree) does not compensate the overhead of sorting and filtering.

**Skewed Accesses.** Figure 8 shows the behavior for the same set of experiments but with a skewed key distribution with a skew factor of 0.2 where 80% of the accesses are concentrated on 20% of the keys. Note that PiBench guarantees inserts and deletes always succeed by only generating requests to non-existing and existing keys, respectively. As a result, skew factor does not influence the generated keys. Since a skewed workload accesses a small subset

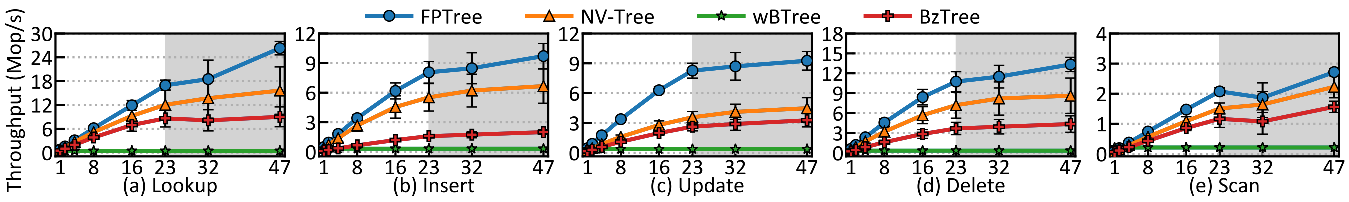

**Figure 11:** Throughput under uniform distribution. FPTree and NV-Tree leverage DRAM and perform generally better than pure PM trees (BzTree and wBTree). All the trees maintain their throughput with hyperthreading (beyond 23 threads). wBTree's single-thread throughput is shown for reference as it does not support concurrency.

of keys multiple times, only the first insert/delete for a given key would succeed and all the subsequent insert/delete operations for the same key would simply be a lookup. Therefore we omit these operations under skewed workloads. As Figure 8 shows, the skewed distribution does enable a better use of CPU caches, which translates directly to higher throughput, and less DRAM and PM reads, while DRAM and PM writes remain very similar. We further vary the skewness of the workload from 0.1 (10% of keys accessed by 90% of requests) to 0.5 in Figures 13(a) and 13(b). As contention level decreases from skew factor 0.1 to 0.5, single-thread throughput drops as a result of more accesses to PM and more cache misses.

## 5.5 Multi-threaded Performance

Now we evaluate the multi-threaded performance of FPTree, NV-Tree, and BzTree. We include wBTree's single-thread performance for reference as it does not support concurrency. In all experiments we first load the trees with 100 million key-values pairs (8-byte keys, 8-byte values), and then measure the run phase consisting of executing 100 million operations split between the worker threads. Since PiBench dedicates one thread to collecting statistics, we scale the number of worker threads until 23, and test with 32 and 47 threads to show the behavior of the trees under hyperthreading.

**Individual operations.** Figure 11 depicts the throughput under uniform distribution. It shows a similar trend to the single-threaded experiments. All the evaluated trees scale as expected for lookup, insert, update, delete and scan operations using 1–23 threads (no hyperthreading). With hyperthreading (shaded areas in Figure 11), all trees maintain or slightly improve compared to using 23 threads. In particular, FPTree is able to leverage hyperthreading significantly better than other trees in lookup operation.

Figure 12 shows the throughput of individual operations under the skewed distribution (skew factor 0.2, we discuss results under other skew factors later). As mentioned previously, we omit insert and delete operations for skewed workloads. The results here showed similar pattern to the ones with the uniform workload: all trees exhibit higher throughput and largely scale under all operations, except BzTree and FPTree's update operation, which respectively scales up to 8 and 16 threads and performs worse as we add more threads. There are two mains reasons for BzTree's behavior. First, because of the use of PMwCAS, a memory word may store a pointer or actual value. Each PM read is instrumented to check the type of the word value, adding additional overhead. Second (and more importantly), the update operation employs an optimistic approach that retries a PMwCAS until success; it is well known that optimistic approaches are vulnerable to high contention. FPTree does not scale beyond 16 threads for a similar reason: it uses HTM (Intel TSX which is an optimistic approach) for traversing the inner nodes and acquiring leaf-level locks. A skewed workload will incur more conflicts at the leaf level, hence more HTM aborts and lower throughput.

For lookup operations, as we vary the skew factor from 0.1 to 0.5 in Figure 13(c), we see the similar overall trend of dropping

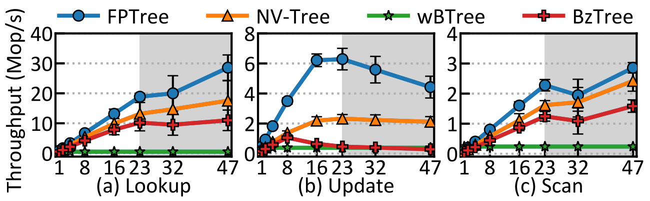

**Figure 12:** Throughput under the skewed distribution (skew factor 0.2). FPTree and BzTree do not scale for updates due to their use of optimistic concurrency control.

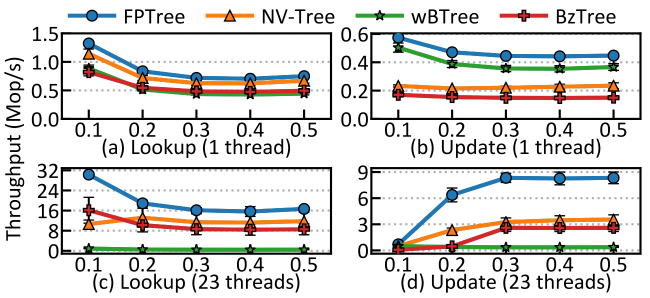

**Figure 13:** Throughput under varying skew factors with one (a–b) and 23 (c–d) threads. Higher skew factor means lower contention.

throughput as the single-thread case for FPTree and BzTree, as lower contention (e.g., skew factor 0.5) leads to accesses to more keys and therefore more cache misses and PM accesses. NV-Tree does not show obvious change when we ease contention. We attribute this behavior to the fact that it needs to acquire node locks even for read-only workloads, causing extra inter-core communications and traffic on the memory bus which is often unscalable for read-only workloads on multicores [36, 41]. Update operations exhibit a different trend in Figure 13(d), as we ease the contention throughput increases, although lower contention leads to larger PM footprint in general, as Figure 13(b) shows. These results highlight two factors that affect performance under skewed workloads: (1) the amount of PM accesses and (2) contention level. Both factors impact performance, and as we add more concurrent threads, contention takes over to become the major factor, contrasting with the single-thread case where PM footprint is the dominating factor.

**Mixed workloads.** Now we examine the trees using three more realistic, mixed workloads under uniform and skewed distributions:

- Read-heavy: 90% lookups and 10% updates;
- Balanced: 50% lookups and 50% updates;
- Write-heavy: 10% lookups and 90% updates.

As shown by Figure 14(a–c), FPTree, NV-Tree and BzTree all scale under the uniform distribution, with FPTree performing more

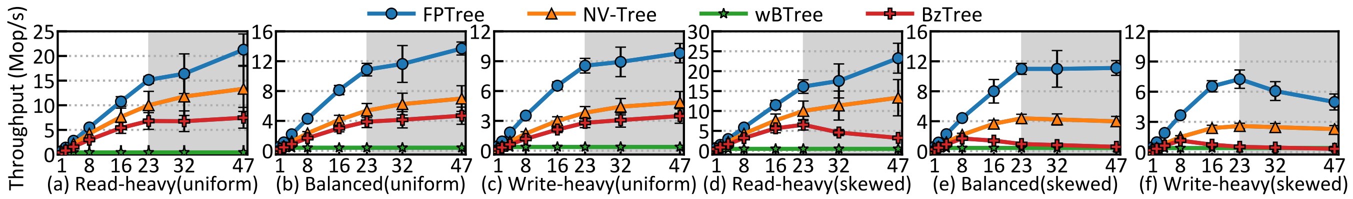

**Figure 14:** Throughput under a varying number of threads and mixed workloads with uniform and skewed (factor 0.2) distributions.

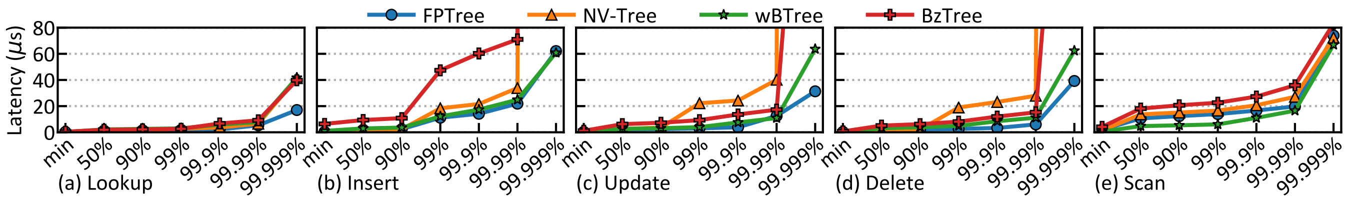

**Figure 15:** Latency at different percentiles for each tree and operation under uniform distribution with a single thread.

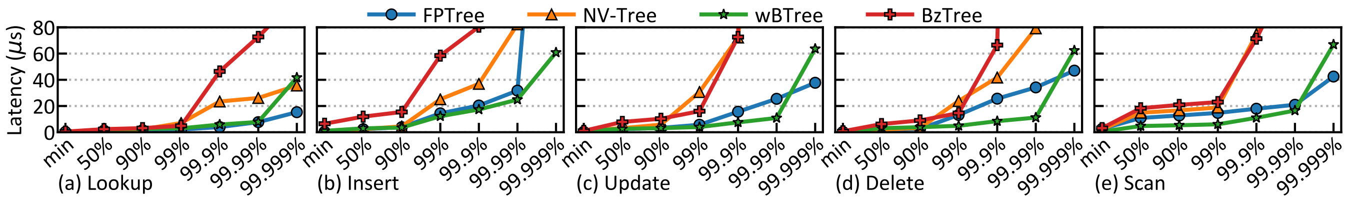

**Figure 16:** Latency at different percentiles for each tree and operation under uniform distribution with 23 threads.

than 2× better than BzTree, due to its various optimizations (leveraging DRAM, fingerprinting). Under the skewed distribution, however, none of the trees are perfect: they either do not scale across the entire horizontal axis or do not obtain high performance in all the workloads. BzTree exhibits more performance drop as we add the percentage of updates to the workload, shown by Figure 14(d–f). Our profiling results show that under the skewed workload, most CPU cycles were spent on retrying the PMwCAS operation needed in each update operation which fails more often as we add more threads and more update percentage in the workload. FPTree performs well under the read-heavy and balanced workloads, but fails to scale for the write-heavy workload, due to high HTM transaction abort rate under high contention. Although NV-Tree "scales" in all cases, it did not achieve the best performance, with up to 3× slower than the best performer, FPTree, which again does not always scale in all workloads.

## 5.6 Tail Latency

Tail latency is another important metric that impacts end-to-end performance. Measuring tail latency is not as straightforward as measuring throughput as it adds significant overhead to the tested operations to accurately store the latency of each operation. Moreover, tail latency should be considered in the context of the throughput the index achieves, since many designs trade tail latency for throughput. As an example, tree *A* might have 2× higher tail latencies than tree *B* but achieve 10× higher throughput. Therefore, we take a sampling approach that samples 10% of the requests uniform randomly; we have experimented with different sampling rates up to 100%, and found 10% to be representative of the behavior while introducing less overhead. We run experiments under uniform distribution to rule out caching effects and achieve more stable access patterns. We also experimented with more skewed distributions, but did not obtain more additional insights and therefore omit them here. We consider two scenarios: single-thread and 23 threads. The former allows a better evaluation of tail latency in isolation, since at this point the throughput of all trees is the most similar. The latter shows the latency behavior under concurrent accesses without hyper-threading, which would introduce additional disturbance.

Figure 15 shows single-thread tail latencies under uniform distribution. For lookups, the minimum latencies for FPTree, NV-Tree and wBTree are all below 0.3$\mu$s, while the number for BzTree is 0.46$\mu$s. As we analyze different latencies percentiles, all trees' latency increases significantly at 99.999 percentile. With modifications to the tree structure, in insert, update and delete operations we observe BzTree and NV-Tree having higher latency than the other two trees. As Figure 15(b) shows, BzTree exhibits over ∼50$\mu$s latency starting from 99 percentile, due to its CoW policy for inner nodes. NV-Tree needs to rebuild inner nodes when split happens in its parent-to-leaf level. However, its use of DRAM for inner nodes helped reduce latency, whereas BzTree is pure PM, putting much pressure on the PM allocator to conduct copy-on-write during splits. Update and delete operations exhibit similar trends in Figures 15(c) and 15(d), with NV-Tree showing the highest tail latency starting from 99 percentile. NV-Tree's behavior for update and delete operations follows its insert operation's behavior, because it handles updates as inserts followed by deletions, as we mentioned in Section 5.4. Scan latency in Figure 15(e) in general shows higher latency than lookup because scan operations themselves are more expensive than point lookups.

Figure 16 repeats the experiment with 23 threads. As wBTree does not support concurrency, we show its single-thread latency numbers for reference. All the other trees exhibit similar but more steep trend compared to the single threaded cases. NV-Tree's latency significantly increased at 99.9 percentile for lookup, since with more concurrent threads its locking mechanism starts to show more overhead. BzTree also had earlier increases at 99 percentile as using more threads presents more physical level contention at the device and memory controller levels [23].

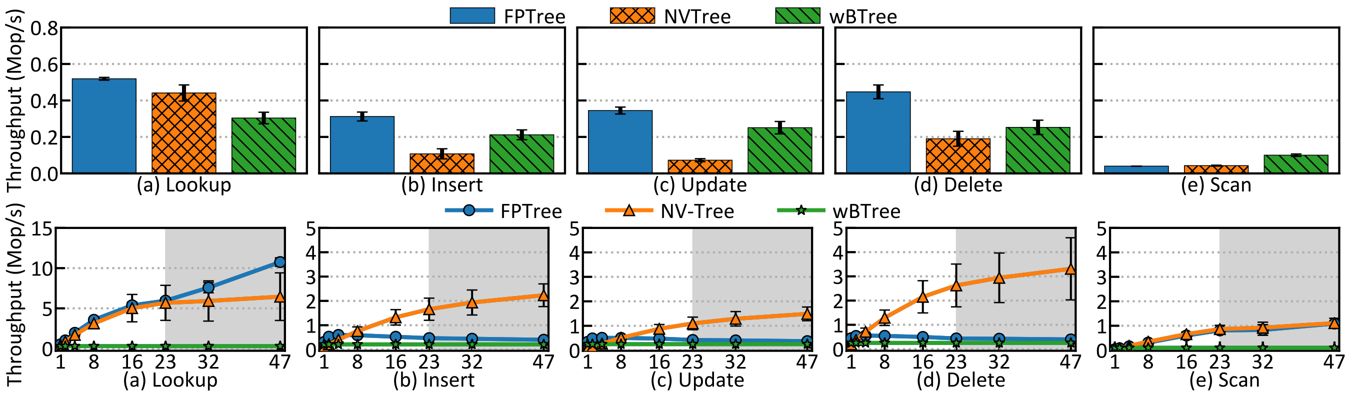

**Figure 17:** Throughput under a single-thread (top) and multiple threads (bottom) with 20-byte keys and 100-byte values.

## 5.7 Long Keys and Values

We run experiments to verify the impact of longer keys and values which are often the case in real-world scenarios. We show the results with 20-byte keys and 100-byte values. The original experiments of BzTree assumed 8-byte values/pointers and storing longer payloads inline was only discussed in the paper [6]. A simple solution would be to delegate the space management and allocate values using PMDK and use pointers to these values. However, in such case, PMDK dominates the performance and hides the behavior of BzTree. Since other trees support native space management, we omit BzTree for fairness. Figure 17 shows the throughput for single thread (top) and multi-thread (bottom) scenarios. As expected, most operations run slower than their counterparts with 8-byte keys and values. The surprise is that FPTree does not scale for insert/update/delete. Through profiling, we found that larger records prevent the use of HTM by triggering a lot of cache evictions and thus transaction aborts. The fallback behavior is to acquire a exclusive global mutex in these cases, which prevents it from properly scaling.

## 5.8 Impact of PM Programming Model

PM indexes face extra challenges in handling persistence, recovery and concurrency, which can be resolved using a sound programming model enforced by some PM programming library (PMDK in our work). Specifically, this boils down to the use of persistent pointers, alignment and a PM-aware allocator, which as we show next, incur space amplification and performance overheads.

**Space amplification.** Similar to in-DRAM OLTP indexes [48], PM indexes may occupy a significant amount of memory (PM and/or DRAM), due to various design decisions to optimize performance (e.g., alignment) and conform to the required PM programming paradigm, in particular the use of 16-byte persistent pointers [19]. We quantify this effect in Figure 18 by plotting the amount of memory consumed by each tree. We insert 100 million records of 8-byte keys and 8-byte values; this corresponds to ∼1.5 GB of raw data. Any consumption beyond this amount would be the metadata, extra alignment or other allocations (e.g., during a split) needed by the tree. We use statistics from jemalloc for DRAM (`stats.allocated`), and the `pmempool` tool for PM. The PM consumption is precise, and DRAM consumption is an upper bound of the real consumption, as jemalloc also records other allocations made by PiBench itself.

As shown by Figure 18, all the trees in fact use more than 50% of the space needed for raw data; NV-Tree/BzTree use respectively ∼2×/10× the raw data size. This is partially due to the relatively small key/value sizes used (8-byte) and the alignment requirement (typically 8-byte) in all the trees. Although both FPTree and NV-

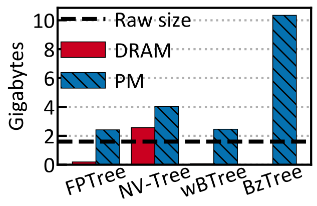

**Figure 18:** Memory consumption after inserting 100M records of 8-byte keys and 8-byte values.

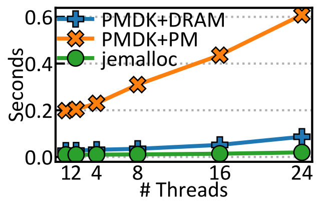

**Figure 19:** Time for allocating 1024 blocks (1KB each) with different allocators.

Tree place inner nodes in DRAM, NV-Tree requires more DRAM because inner nodes are rebuilt sparsely to amortize rebuild cost. This means that it will have a different PM–DRAM ratio depending on the fill ratio of its inner nodes. BzTree and wBTree consume negligible amount of DRAM as they are pure PM-based. Among all the trees, BzTree's memory consumption is cumulative (of all nodes ever created) and the highest, due to its use of CoW for inner nodes. We note, however, that this is the worst case for BzTree, and in realistic workloads with fewer inserts, inner nodes will not change as often, which should result in lower PM consumption.

**PM allocation overhead.** Compared to their DRAM counterparts, persistent allocators need to issue cacheline flush instructions, handle recovery and run on slower PM. To understand their behavior, we run an experiment using jemalloc on DRAM and PMDK allocator (which is based on jemalloc) on DRAM and PM. Each thread issues 1024 allocation requests, each of which allocates 1KB of memory from the allocator. Figure 19 shows the time needed to finish the test. As we increase the number of threads, no allocator scales, and due to the extra complexity associated with PMDK allocator (e.g., the use of cacheline flush instructions and fences), PMDK allocator is 2.9–4.4× slower than jemalloc on DRAM. On Optane DCPMM, the PMDK allocator can be up to ∼8× slower than itself running on DRAM. These results signify the high cost of PM allocators and indicate that PM data structures should carefully handle their interactions with PM allocators.

In tree structures, the insert operation interacts with allocators the most, and we observed non-trivial allocation overhead in all the evaluated trees. In particular, we found that BzTree spends more than ∼41% of CPU cycles on PM allocation in insert operations. Compared to other trees, its use of CoW adds more burden on the PM allocator because an inner node cannot be updated in-place when a new key is added to it. We observed similar trends in other trees; we omit the details here due to space limitation.

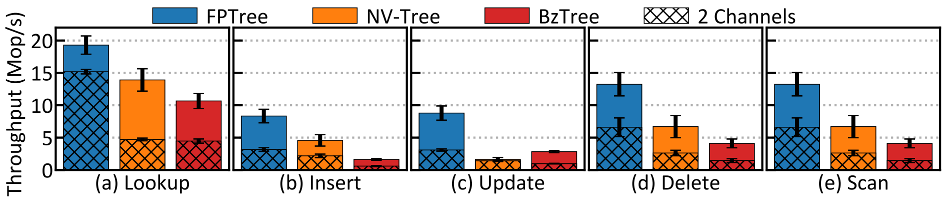

**Figure 20:** Throughput when using six and two (shaded areas) PM DIMMs under 23 threads and uniform distribution. With two channels, all the trees are bottlenecked by PM bandwidth.

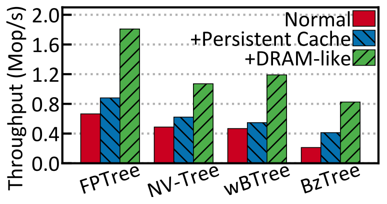

**Figure 21:** Impact of persistent CPU caches and DRAM-like performance.

## 5.9 Recovery

An important aspect of persistent data structures is their ability to recover consistently and (near) instantly after a failure or clean shutdown. We test recovery time by loading in each tree a fixed number of records and then kill the process. Table 3 illustrates the time in seconds for each tree to recover from a crash after loading 50 million and 100 million records. wBTree, FPTree and NV-Tree enforce consistency of each single operation, so recovery consists basically of rebuilding transient data. Both FPTree and NV-Tree place inner nodes on DRAM and thus these must be rebuilt upon recovery. As expected, the time for rebuilding inner nodes after inserting 100 million records is about 2× the time with 50 million records (recovery time scales linearly as all leaf nodes must be read).

wBTree and BzTree reside fully in PM. Therefore, upon recovery they simply need to open an existing PM pool and retrieve the root object. From the root object all the remaining objects allocated in the pool can be discovered and reached. BzTree relies on PMwCAS to always transform the tree from one consistent state to another, without needing a customized recovery procedure. After opening the existing pool, BzTree delegates its recovery process to PMwCAS, which completes its own recovery phase by rolling forward or backward PMwCAS operations that were in-progress when the crash happened. This translates into scanning the PMwCAS descriptor pool [42] which is fixed-sized (100k in our experiments). Moreover, the amount of in-progress PMwCAS operations at any point in time is bounded by the number of concurrent threads. Therefore, we see a very small difference in recovery time under different initial sizes.

These result show that only PM-only trees are able to recovery near instantly (sub-second recovery time), at the price of lower runtime performance; placing more components in DRAM may improve runtime performance at the cost of longer recovery time. Nevertheless, we note that the recovery time of hybrid trees could be improved, in case of clean shutdowns, by spilling inner nodes to PM, then copying them back to DRAM upon recovery.

## 5.10 Impact of PM Bandwidth

As mentioned in Section 5.1, our test machine is fully populated with six DCPMMs, which gives us enough bandwidth for all the trees to conduct multi-threaded experiments. We also observe that, under multiple threads, operating these trees consume a significant portion of the available PM bandwidth. We conducted additional experiments using only two DCPMMs (i.e., utilizing two out of the six channels). Figure 20 shows the throughput of each individual operation obtained under 23 threads and the uniform distribution. The hatched bars represent the throughput achieved with two channels, while solid bars represent the throughput achieved with six channels. Note that the bars are overlaid (not stacked). With two channels, the trees were only able to obtain ∼30–70% of the performance that could be obtained with six DCPMMs that utilize all the available memory channels. In fact, we found that with two channels, all trees stop scaling at 16 threads, hitting the PM bandwidth limit, whereas with six channels bandwidth is no longer the key factor affecting scalability, as discussed in Section 5.5. This result underlines the importance of PM bandwidth for PM data structures: DCPMMs usually come with large capacity and system builder may be tempted to use fewer DCPMMs for the same capacity to simplify system design and leave more slots for DRAM which does not scale as well as PM. This will put more pressure on data structures in PM, which must be designed carefully to use the limited bandwidth in case the system is not fully populated, in addition to the fact that PM already offers lower bandwidth and higher latency at the device level.

**Table 3:** Recovery time after 50M and 100M inserts.

| Initial Size | FPTree | NV-Tree | wBTree | BzTree |
|---|---|---|---|---|
| 50M | 1.77s | 4.15s | 0.036s | 0.153s |
| 100M | 3.56s | 8.45s | 0.037s | 0.186s |

## 6. DISCUSSION

Now we summarize the observations and distill insights from our evaluation. We hope they can serve as guidelines in devising indexes on PM. Most of them apply beyond indexes to other data structures. Motivated by our findings, we give a wish list of hardware features that could ease the making of PM-based indexes and data structures.

### 6.1 Observations, Insights and Caveats

**1. The evaluated indexes run factors slower on real PM than originally reported.** The first reason is the lack of a sound PM programming model. Most evaluated trees were prototyped using virtual pointers and a transient memory allocator. FPTree was originally prototyped following a programming model similar to that of PMDK, but using a closed-source PM allocator. By implementing these trees using PMDK's programming model, we discovered that some advocated design decisions, such as copy-on-write, are not ideal (see below). The other reason is imprecise PM emulation: although the original papers had to use emulation based on DRAM before real PM hardware became available, a major aspect that they failed to capture was the scarcity of PM bandwidth.

**2. PM Bandwidth is a scarce resource.** In the best case of a fully populated system, PM's bandwidth for sequential read, sequential write and random write are ∼3×, ∼11× and ∼14× lower than that of DRAM, respectively. These factors are even larger when fewer DCPMMs are used. Bandwidth was never an issue for DRAM trees, but as we have highlighted, it might well become one with PM.

**3. PM allocations are very expensive.** PM allocations are orders of magnitude slower than DRAM allocations as Section 5.8 shows, making them a noticeable performance differentiator that was not accounted for. This holds not only for the PMDK allocator, but for all proposed PM allocators, as they all have to persist metadata to ensure proper transfer of ownership [31]. To reduce allocation overheads, in-place updates and bulk allocation should be favored.

**4. Leveraging DRAM is desirable for high performance but may trade off recovery time and cost of ownership.** Given PM's limited bandwidth, this is especially important for concurrent data structures [23, 46], but recovery time can become longer if much needs to be rebuilt on DRAM upon recovery. Cost of ownership will also become higher as DRAM is more expensive.
**5. Fingerprinting is effective in speeding up point queries.** Our results showed that fingerprinting significantly reduces PM accesses for point operations (lookup, insert, update, and delete), thereby achieving higher performance and levels of concurrency.
**6. Indirection slot arrays significantly speed up range queries.** Our results clearly show that indirection slot arrays is the go-to technique when range scans are required. It implicitly stores the order of the records such that, despite the necessary unsorted leaf structure to achieve fast failure atomicity, range scans require neither scanning the whole leaf nor reconstructing the order of the records.
**7. Copy-on-Write is a bad fit for PM.** The reasons are (1) CoW amplifies the number of required expensive PM allocations as shown in Section 5.8, and (2) it consumes additional PM bandwidth, a limited resource as demonstrated in this work.
**8. Applicability to NVDIMM.** Although we focused on Optane DCPMM, techniques for reducing writes and flushes, e.g., unsorted nodes and fingerprinting, also apply to NVDIMM. On NVDIMM, PM allocations will also be expensive due to the need of flushes and fences, making CoW a bad fit. Bandwidth will be less of a problem, unless the system does not leverage enough memory channels.

### 6.2 Persistent Memory Wish List

PM is still at an early stage and yet to become mainstream – we believe it will. This work enabled us to identify two areas where improvements of the current PM could have a major impact.

**Persistent CPU caches.** Modern CPUs rely on sophisticated, fast volatile caches for good performance. This introduces the main challenge of carefully flushing cachelines to PM while trying to reduce the amount of flushes and PM accesses. We consider that enabling CPU caches to become persistent (e.g., by protecting them against power failures with a capacitor) is the natural next step to simplify software development and increase performance [22, 40]. Then, applications can completely relinquish the use of instructions such as `CLFLUSHOPT` and `CLWB`. However, guaranteeing the ordering of writes using `SFENCE` may still be required.

**DRAM-like PM devices.** A second advancement would be approaching the performance characteristics of DRAM. NVDIMMs (DRAM backed by flash and supercapacitor [5, 39]) already offer DRAM performance, but its high cost and DRAM's scalability issues make it prohibitive in large scale. Devices based on new materials are much cheaper but still lag behind DRAM in terms of performance. This gap might be closed by reducing the cost of flash-based PM or enhancing the cheaper alternatives (e.g., via innovations in materials or more sophisticated caching mechanisms).

Figure 21 shows the potential impact of these advancements. We emulate persistent CPU caches (`+Persistent Cache`) by removing all cacheline flushes from the code path, and emulate fast PM (`+DRAM-like`) by placing the PM pool in a DRAM-backed file system (`tmpfs`). While persistent CPU caches improve throughput by 1.32/1.27/1.17/1.94× for FPTree/NV-Tree/wBTree/BzTree respectively, the main benefit is probably in terms of simplifying the programming model which will also lead to fewer bugs and savings in development and code maintenance costs. The biggest absolute gains are achieved by increasing the raw device performance, which further improves the throughput by a factor of 2.05/1.72/2.17/2.00 for FPTree/NV-Tree/wBTree/BzTree respectively. This shows that indexes are highly sensitive to device latency and bandwidth.

## 7. RELATED WORK

**Tree structures.** CDDS $B^+$-Tree [38] was one of the early persistent and concurrent $B^+$-Tree structures. It relies on versioning for failure atomicity and visibility. Its scalability may suffer from using a global version number. RNTree [28] leverages HTM and a new slot array approach to reduce persistence and sorting overheads. Hwang et al. [18] proposed a B-link-tree for PM using failure-atomic in-place shift and rebalance. The former shifts sorted entries in nodes in a data-dependent way to prevent inconsistencies. The latter leverages B-link-tree sibling pointers to perform node splits without logging.

While our focus is on B+-trees, for completeness we cover radix trees here. Lee et al. [25] proposed three variants of PM-aware radix trees based on path-compressed radix tree [29] and ART [26]. The authors use a combination of bitmaps, indirection slot arrays, and 8-byte pointers, all of which can be updated in a failure atomic way.

**Hash tables and hybrid structures.** NVC-Hashmap [34] was one of the early persistent and concurrent hash maps. Level hashing [50] bases on Cuckoo hashing and minimizes PM writes and efficiently handles hash collisions (without providing data durability). Nam et al. [30] proposed a PM-based extendible hashing scheme, which uses a level of indirection to limit the number of cache misses during hash probing. HiKV [44] is a hybrid key-value store that uses a partitioned persistent hash index for primary data, and a transient global $B^+$-Tree for range queries. PMDK [19] also includes several examples: a linked list, hash table, binary search tree, and `pmemkv`, a key-value store partially based on FPTree.

**PM primitives.** Building persistent indexes requires the use of PM programming libraries. PMDK [19] has been the most popular one and provides a set of primitives for handling durability, PM space management, persistent pointers, allocation and logging. PMwCAS [42] fills the gap of lock-free concurrency with an easy-to-use multi-word interface, and is used by the BzTree. Instead of logging, PMwCAS employs a "dirty bit" design where each written memory word is marked as "dirty" before it is flushed to PM; threads that see the dirty bit set will flush the word and atomically unset the dirty bit before reading the actual word. This ensures threads only read "committed" data. Log-free data structures [13] use a similar technique to avoid logging. They use single-word atomics and techniques such as buffered writes, while PMwCAS provides a multi-word abstraction to ease lock-free programming on PM.

## 8. CONCLUSION

In this paper, we revisited PM-based range indexes on the newly released Intel Optane DC Persistent Memory. Focusing on $B^+$-Tree-like structures, we carefully selected four representative indexes that cover a wide range of solutions to the challenges raised by PM. To fairly benchmark the trees and enable reproducibility, we devised PiBench, a framework for benchmarking indexes on PM. PiBench can work with any index that supports the common operations.

Using PiBench, we unveiled important, non-trivial insights, such as the impacts of programming model and limited bandwidth, and how copy-on-write can be ill-suited due to PM's limited bandwidth and allocator overheads. We also pinpointed designs that allowed the evaluated data structures to excel for specific workloads. These techniques are mostly orthogonal to each other, and together with our insights, we hope they can serve as building blocks for designing future PM data structures. Finally, we quantified the performance gains that would stem from persistent CPU caches and DRAM-like PM performance. We found performance is mainly dictated by PM's latency and bandwidth characteristics, and the slowdown incurred by cacheline flushes is relatively limited in comparison.

## 9. REFERENCES

[1] Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt .

[2] Ext4 Filesystem. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt .

[3] Intel 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual .

[4] The SGI XFS Filesystem. https://www.kernel.org/doc/Documentation/filesystems/xfs.txt .

[5] AgigaTech. AgigaTech Non-Volatile RAM. 2017. http://www.agigatech.com/nvram.php.

[6] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB*, 11(5):553–565, 2018.

[7] R. Bayer. Binary B-Trees for Virtual Memory. In *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '71, pages 219–235, 1971.

[8] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 677–694, 2016.

[9] S. Chen, P. B. Gibbons, and S. Nath. Rethinking Database Algorithms for Phase Change Memory. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 21–31, 2011.

[10] S. Chen and Q. Jin. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB*, 8(7):786–797, 2015.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[12] R. Crooke and M. Durcan. A Revolutionary Breakthrough in Memory Technology. *3D XPoint Launch Keynote*, 2015.

[13] T. David, A. Dragojević, R. Guerraoui, and I. Zablotchi. Log-free Concurrent Data Structures. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC '18, pages 373–385, 2018.

[14] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, 2014.

[15] J. Evans. A Scalable Concurrent malloc (3) Implementation for FreeBSD. In *Proceedings of the BSDCan Conference*, 2006.

[16] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. *ACM SIGMOD Record*, 23(2):243–252, 1994.

[17] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. *IEEE International Electron Devices Meeting (IEDM)*, pages 459–462, 2005.

[18] D. Hwang, W.-H. Kim, Y. Won, and B. Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, 2018.

[19] Intel. Persistent Memory Development Kit. 2018. http://pmem.io/pmdk.

[20] Intel Corporation. Intel Optane DC Persistent Memory Readies for Widespread Deployment. 2018. https://newsroom.intel.com/news/intel-optane-dc-persistent-memory-readies-widespread-deployment/.

[21] Intel Corporation et al. Processor Counter Monitor. 2019. https://github.com/opcm/pcm.

[22] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, 2016.

[23] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, et al. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714*, 2019.

[24] D. E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional, 2014. Section 6.4: Hashing.

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

[26] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing For Main-Memory Databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49. IEEE, 2013.

[27] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 302–313, 2013.

[28] M. Liu, J. Xing, K. Chen, and Y. Wu. Building Scalable NVM-based B+tree with HTM. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, pages 101:1–101:10, 2019.

[29] D. R. Morrison. PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[30] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, 2019.

[31] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems. *PVLDB*, 10(11):1166–1177, 2017.

[32] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. 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*, SIGMOD '16, pages 371–386. ACM, 2016.

[33] A. Rudoff. Deprecating the PCOMMIT Instruction. *Intel Software Documentation*, 2016. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[34] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. NVC-Hashmap: A persistent and Concurrent Hashmap For Non-Volatile Memories. In *Proceedings of the 3rd VLDB*

*Workshop on In-Memory Data Mangement and Analytics*, page 4. ACM, 2015.

[35] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[36] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, 2013.

[37] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1541–1555. ACM, 2018.

[38] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, pages 61–75, 2011.

[39] Viking Technology. DDR4 NVDIMM. 2017. http://www.vikingtechnology.com/products/nvdimm/ddr4-nvdimm/.

[40] T. Wang and R. Johnson. Scalable Logging Through Emerging Non-Volatile Memory. *PVLDB*, 7(10):865–876, 2014.

[41] T. Wang and H. Kimura. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *PVLDB*, 10(2):49–60, 2016.

[42] T. Wang, J. Levandoski, and P. Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.

[43] H. S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson. Phase Change Memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[44] F. Xia, D. Jiang, J. Xiong, and N. Sun. HiKV: A Hybrid Index Key-value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC '17, pages 349–362, 2017.

[45] J. Xu and S. Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[46] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory, 2019.

[47] J. Yang, Q. Wei, C. Wang, C. Chen, K. L. Yong, and B. He. NV-Tree: A Consistent and Workload-adaptive Tree Structure for Non-Volatile Memory. *IEEE Transactions on Computers*, 65(7):2169–2183, July 2016.

[48] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1567–1581. ACM, 2016.

[49] Y. Zhang and S. Swanson. A Study of Application Performance with Non-Volatile Main Memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2015.

[50] P. Zuo, Y. Hua, and J. Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.