uTree: a Persistent B+-Tree with Low Tail Latency

Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, Jiwu Shu
Tsinghua University
Corresponding Author: Jiwu Shu (shujw@tsinghua.edu.cn)

ABSTRACT

Tail latency is a critical design issue in recent storage systems. B$^+$-tree, as a fundamental building block in storage systems, incurs high tail latency, especially when placed in persistent memory (PM). Our empirical study specifies two factors that lead to such latency spikes: (*i*) the internal structural refinement operations (i.e., split, merge, and balance), and (*ii*) the interference between concurrent operations. The problem is even worse when high concurrency meets with the low write bandwidth of persistent memory.

In this paper, we propose a B$^+$-tree variant named μTree. It incorporates a shadow list-based layer to the leaf nodes of a B$^+$-tree to gain benefits from both list and tree data structures. The list layer in PM is exempt from the structural refinement operations since list nodes in the list layer own separate PM spaces, which are organized in an *element-based* way. Meanwhile, μTree still gains the locality benefit from the tree-based nodes. To alleviate the interference overhead, μTree coordinates the concurrency control between the tree and list layer, which moves the slow PM accesses out of the critical path. We compare μTree to state-of-the-art designs of PM-aware B$^+$-tree indices under both YCSB workload and real-world applications. μTree achieves a 99th percentile latency that is one order of magnitude lower and 2.8 - 4.7 times higher throughput.

PVLDB Reference Format:
Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, Jiwu Shu. μTree: a Persistent B$^+$-Tree with Low Tail Latency. *PVLDB*, 13(11): 2634-2648, 2020.
DOI: https://doi.org/10.14778/3407790.3407850

1. INTRODUCTION

Web-scale online data-intensive applications such as social applications and e-commerce bring a new performance focus, i.e., tail latency, to modern warehouse-scale data centers, such as the 99th percentile (99p) of request latencies [4]. Tail latency is particularly important since applications often exhibit high fan-out queries, whose overall latency is determined by the response time of the slowest reply. To this end, a number of recent works proposed different ways to reduce tail latency, such as distributing all critical data in the memory [18, 19, 52, 54], scheduling CPU cores at finer granularity [27, 44, 45, 47], and designing tail latency-aware caching policies [9].

In this paper, we target at reducing the tail latency of tree-based index structures (e.g., B$^+$-tree), which are widely adopted in storage systems with simple interfaces (e.g., `Get`, `Put`, `Del`, and `Scan`, etc.). Emerging persistent memory (PM), such as the Intel Optane DC persistent memory (abbreviated as Optane DCPMM or Optane in the rest of paper) [41], provides high performance and durability features, making it an attractive medium for building efficient index structures. In the past decade, researchers have designed many PM-based B$^+$-tree indices, such as FAST&FAIR [23], bzTree [6], DPTree [59], RNTree [39] FPTree [43], NV-Tree [57], wB$^+$-tree [12], CDDS-Tree [53], etc. All of them mainly target at improving the throughput (e.g., Ops) and focus on reducing the write cost when updating the tree structure. This is based on the fact that PM typically has asymmetric read/write performance. Our evaluation shows that Optane DCPMM has a read bandwidth of 39 GB/s (six Optane modules, 4KB sequential access), which is much higher than the write bandwidth (13.2 GB/s).

Despite various optimizations on throughput, we demonstrate that tail latencies in state-of-the-art PM-aware B$^+$-tree are quite poor, especially under write-intensive and skewed workloads. Our evaluation shows that FAST&FAIR, a well-optimized persistent B$^+$-tree, exhibits almost 60μs of 99th percentile latency as its throughput peaks, which is 600 times higher than the PM write latency. The tail latency can be further amplified when a B$^+$-tree is accessed multiple times to handle a single application request, which is common in real-world applications [15].

We specify two reasons that account for such a wide spectrum of latency distribution. First, B$^+$-tree introduces internal structural refinement operations (SROs). In a B$^+$-tree, each tree node contains multiple entries and is organized in an *array-based* way (i.e., the entries occupy a contiguous memory space). Therefore, to provide high search performance, one must shift the entries inside each tree node to keep them sorted. Besides, a tree structure also needs to merge or split tree nodes to keep *balanced*, limiting the search complexity into log(N) (i.e., structural modification operation, SMO). Note that SROs are more heavy-weight since they involve modifying more data in PM. As a result, the time consumed to process each `Put` or `Del` opera-

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407850

tion is dynamically changed, exhibiting unpredictable performance. Recent works have proposed different ways to reduce the overhead caused by SROs: wB$^+$-Tree [12] and FPTree [43] enable unsorted entries inside each tree node; FPTree and NV-Tree [57] place inner tree nodes directly in DRAM. However, the SRO overhead in PM still cannot be dismissed completely.

Second, concurrent threads are likely to interfere with each other, even if they access independent data items in a B$^+$-tree. For example, in FAST&FAIR, the entries in the same tree node share the same lock, and all accesses to this node are serialized, even if they update different entries. Things become even more serious when an SRO is holding the lock, a situation where a simple `Get` may experience additional latency. What's more, our evaluation reveals that when there are more concurrent threads, the latency of an update to PM can be as high as more than 1μs, because PM has limited write bandwidth. In such a case, the tail latency is further exacerbated.

This paper proposes a B$^+$-tree variant named μTree to reduce tail latency. We introduce the singly linked-list to leaf nodes of a B$^+$-tree as a shadow layer. The linked-list is organized in an *element-based* way: The list nodes of different keys own separate memory spaces, and the data items can be kept sorted by simply manipulating the sibling pointers. Therefore, the list layer is exempt from the SRO overhead as in the *array-based* management. Following a DRAM/PM hybrid data placement, the list layer is placed in PM, and other nodes, including tree-leaf nodes, are placed in DRAM. With this design, μTree gains the respective advantages of each layer: Clients search data items via the tree nodes, with O(logN) of complexity and higher cache locality; A client updates PM via the list layer, eliminating the SRO overhead. Note that the tree layer in DRAM still causes SRO overhead, however, it has less impact on the tail latency since DRAM is more efficient. With the list layer, μTree can also rebuild the volatile tree nodes once the system crashes.

To alleviate the interference overhead, μTree proposes the *coordinated concurrency control* technique by exploiting the element-grained concurrency of the shadow list layer. Specifically, μTree inserts a list node in the list layer by using a sequence of atomic operations leveraging its lock-free feature, and speculatively persists them without acquiring the lock in the B$^+$-tree. This is achievable since the newly inserted list node is not visible until a tree-leaf node pointing to it is inserted in the tree-leaf node (i.e., array layer). Once a conflict occurs and causes inconsistencies between the list and array layer, μTree always retries from the array layer to reload the newest state of the list layer. In this way, μTree supports more fine-grained concurrency control and the slow PM accesses are moved out of the critical path. We also encapsulate extra access control bits in each list node to isolate incomplete modifications, so as to deal with the `Put-Get` and `Put-Del` conflicts. We make the following contributions:

- We perform an in-depth analysis of the tail latency problem in persistent B$^+$-trees and specify two root causes.
- We propose μTree by introducing a shadow list-based layer to leaf nodes of a B$^+$-tree. We then reduce the interference overhead with coordinated concurrency control.
- We implement μTree and our evaluation shows that μTree achieves one order of magnitude lower 99th percentile latencies. It also achieves a peak throughput that is 2.8 - 4.7× higher.

2. BACKGROUND AND MOTIVATION

In this section, we first describe the PM programming model. Several past systems that improve the throughput of persistent B$^+$-trees are then introduced. Finally, we empirically analyze the tail latency issue in a persistent B$^+$-tree.

2.1 Persistent Memory and The New Programming Model

Persistent memories, such as PCM [29, 33, 34, 48, 51, 58], STT-RAM [5,31] and ReRAM [3], have DRAM-comparable access latency, byte addressability and data persistency. Intel's Optane DCPMM, the first PM product, was released in April 2019 [41]. It is attached to the memory bus and accessed via CPU `load/store` instructions. Optane DCPMM has *asymmetric read and write performance*: With six Optane DIMMs, the overall read bandwidth can reach 39 GB/s, while the write bandwidth is merely 13.2 GB/s.

Programming in PM is quite different. CPU issues writes to PM in 8-byte failure-atomic units, which is smaller than that of HDDs or SSDs (which are 512-B sectors or 4-KB pages). This requires extra techniques (e.g., redo/undo logs) to achieve atomicity. In PMs, these writes are firstly cached in the volatile CPU cache, and are then written back to the PM in an arbitrary order. Guaranteeing the consistency of data requires us to order the writes. Such ordering implies two things: 1) a write should be visible before any other writes (achieved by following a `mfence`), 2) the order of writes to actually reach the PM controller (by explicitly flushing the cache line via `clflush`, `clwb` or `clflushopt`). The small-sized failure-atomic units and the ordering constraints make it more challenging to design consistent B$^+$-Tree in PMs.

2.2 Existing Persistent B$^+$-tree

B$^+$-tree is a fundamental building block in databases, file systems, and other storage systems. Several recent works build B$^+$-tree indices in PM and most of them are devoted to improving the throughput by reducing write costs. We highlight some of them here:

CDDS B-Tree (Consistent and Durable Data Structure B-Tree) [53] is a multi-version B-Tree, and each tree node stores an extra *version* field. When a tree node is updated, a new copy is created and tagged with a new version, without overwriting the old one. In this way, the recoverability and consistency are achieved.

NV-Tree [57] reduces the number of flush operations with two techniques. First, it only places critical data (i.e., leaf nodes) in PM, and the internal nodes are reconstructable and can be kept in an inconsistent state. Second, the leaf nodes are managed with an append-only strategy. Newly inserted entries are always appended at the end of a leaf node, without keeping the entries sorted.

wB$^+$-tree [12] proposes a slot array, which acts as an indirect layer to keep leaf nodes sorted, while the shifting overhead of sorting tree nodes is avoided.

FPTree [43] is similar to NV-Tree, which always appends entries at the end of a tree node and only places leaf nodes in PM. A one-byte key hash of fingerprint is used to reduce the penalty of cache misses in searching. FPTree uses hardware transactional memory (HTM) and mutex lock for concurrency control. Traversing from the tree root to a leaf node is wrapped by HTM. Once the leaf node is found, FPTree locks the whole leaf node before modifying it.

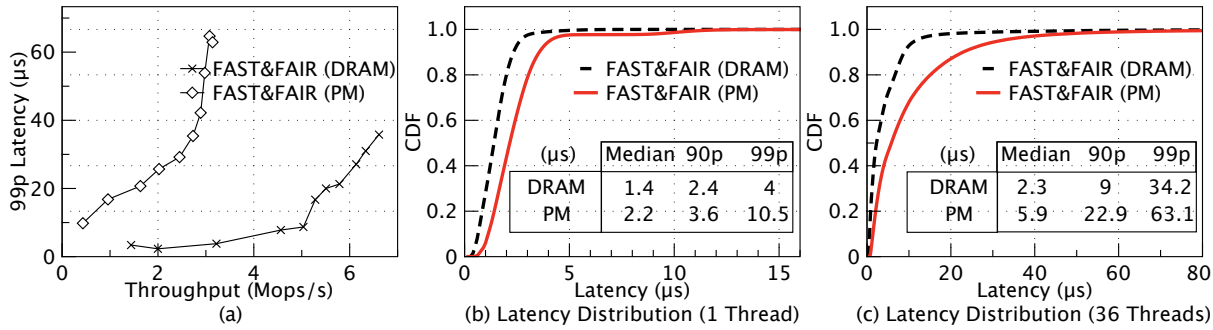


Figure 1: Tail latency analysis by evaluating FAST&FAIR. (FAST&FAIR (DRAM) indicates a volatile version of FAST&FAIR whose data are stored completely in DRAM and flush operations are removed.)

FAST&FAIR [23]. Merge or balance operations involve modifying multiple tree nodes. Hence, the aforementioned index schemes need extra logging mechanisms to guarantee the failure atomicity. FAST&FAIR avoids such logging overhead by updating a B$^+$-tree with relaxed consistency. Any inconsistency states (e.g., duplicated elements) caused by power failure can be specified and fixed. It also supports lock-free reads.

2.3 Tail Latency Analysis

Existing persistent indices mainly target at improving the throughput, either by reducing the persistence overhead (i.e., reducing the number of flush operations), or by reducing the consistency overhead (e.g., FAST&FAIR avoids logging). Except for the throughput goal, this paper is further motivated with another performance metric — tail latency.

To quantitatively understand the tail latency problem, we investigate the latency behavior of FAST&FAIR, a state-of-the-art high-performance persistent index structure. To highlight the effects of PM, we also compare it with a volatile version of FAST&FAIR, which places tree nodes directly in DRAM. Our experiments are conducted on a server with two Intel Xeon Gold 6240M CPUs (36 cores in total), 192 GB of DRAM, and 1.5 TB of Optane DCPMMs (six modules). Detailed platform configuration is shown in §5.

In this part, we evaluate the performance of `Put` operations by using the zipfian key distribution (parameter = 0.99). The size of both keys and values are set to 8 bytes. We bind cores and set the CPU frequency to a fixed value (3.3 GHz) to avoid the DVFS (dynamic voltage and frequency scaling) from impacting the tail latency. We target at the efficiency of `Put` operation since many existing applications are write-intensive and sensitive to the tail latency. For example, the caching system with frequently-changing objects [2], the popular e-commerce platform that processes new transactions at an extremely fast rate [16], and the emerging serverless analytics exchanging short-lived data [30], etc. All of them generate data rapidly and impose increasing demands on the storage system to deliver predictable performance.

Figure 1a reports FAST&FAIR's throughput and 99th percentile latency as we increase the number of threads from 1 to 36. We can observe that its throughput peaks soon as the number of threads increases. At this point, the 99th percentile latency is as high as 60μs, which is 10× higher than its median latency. By placing FAST&FAIR directly in DRAM, we observe that its tail latency is significantly lower: for a target load running at 3 Mops/s, FAST&FAIR (DRAM) shows a 99p latency of only 4μs, 10× lower than that of FAST&FAIR (PM). Besides, FAST&FAIR (DRAM) achieves more than 2× higher peak throughput (around 7 Mops/s). By analyzing latency distributions running at different concurrency levels, we specify two factors that hurt tail latency:

Structural Refinement Operations (SROs). In B$^+$-tree indexes, an insert or delete operation often causes structural refinement operations (e.g., sort and balance). However, SROs incur extra data modifications since they often cause the data movement of multiple entries in one or multiple tree node(s). Considering that PM has lower write bandwidth and updates made in SROs are required to be persisted synchronously, SROs often cause much higher data persistence overhead. Note that SROs happens occasionally and thus only affect a few of `Put/Delete` operations, most operations, instead, still can be finished immediately. Figure 1b reports the latency distribution of Put operations under single-thread execution, which can be used to reveal the effects of SROs. We can observe that FAST&FAIR's 99th percentile latency is 10.5μs, 4.8× higher than the median latency. By comparing the execution path, we find that the Put operations that contain SROs typically appear at the tail of the latency distribution. By placing FAST&FAIR in DRAM, the 99p latency is only 4μs.

To reduce the SRO overhead, NV-Tree [57], wB$^+$-Tree [51] and FPTree [43] allow unsorted keys in the leaf node, by compromising the search/scan efficiency. Recent work also adopts the hybrid DRAM/PM placement. For instance, NV-Tree [57], and FPTree [43] eliminate the persistence overhead of non-leaf nodes by introducing *selective persistence* or simply placing them in DRAM. However, these variants still suffer from sort and balance overhead in leaf nodes, which contributes to the main source of overhead.

Interference Overhead. The dependency between entries makes it difficult to support concurrent accesses to entries inside the same node. Previous work, such as B^{link}-Tree [36], OLFIT [11], and Masstree [40], propose the per-node exclusive lock to serialize updates to the same node, and per-node version to protect readers from observing intermediate states while avoiding the heavy-weight locking overhead. FAST&FAIR also adopts a similar concurrency control strategy (i.e., per-node locking and lock-free reads). From Figure 1c we observe that such concurrency control technique works well in volatile memory. By placing data in PM, however, the high write latency of PM is exposed directly in the critical path, which prolongs the execution time of each update. As a result, other threads accessing the same tree node may experience an extra delay, exacerbating the tail latency problem. To understand such issues micro-

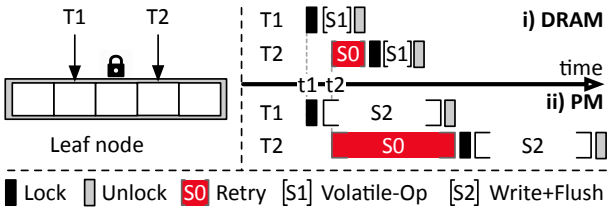


Figure 2: Concurrent insertion to the same leaf node (the right part compares the timing diagram of two cases, in DRAM and PM, respectively).

scopically, we depict the write-write conflicts in DRAM- and PM-based FAST&FAIR (see Figure 2).

As shown in the figure, two threads T_1 and T_2 are updating the same tree node. T_1 updates key K_a at time t_1, and T_2 updates key K_b at time t_2 ($t_1 < t_2$). As we can see, T_2 will retry repeatedly until T_1 completes the update operation and release the lock. This is tolerable in DRAM, as its write latency is low. In PM, however, the high write latency of PM delays the execution of the later write operation, even though the operation itself can complete quickly.

To summarize our discussion so far, existing tree-based persistent indices exhibit high tail latency by nature. Among the factors that impact tail latency, SROs cause the execution time of each operation to change dynamically, and inter-thread interferences further aggravate this problem.

3. DESIGN

Before illustrating the design details, we put forward the following design goals for the μTree.

- *Reduced tail latency and improved peak throughput.*
- *Resilient to power/system failures and strict consistency.*
- *Fast recovery speed.*

To fulfill the above goals, we present the design and implementation of μTree, with techniques including the shadow list layer (§3.1), coordinated concurrency control (§3.2) and fast recovery (§3.3).

3.1 The μTree Structure

μTree is different from B$^+$-tree in the leaf node design. Specifically, the inner nodes of μTree are the same as that of traditional B$^+$-tree, which are placed directly in DRAM. The leaf nodes, as shown in Figure 3, consist of two layers — the array layer and the list layer. Among them, the array layer (i.e., tree-leaf nodes) is stored in DRAM, and only the list layer (i.e., list-leaf nodes) is stored in PM. Each tree-leaf node contains multiple entries (i.e., key-pointer pairs), and each of them points to a list-leaf node. These list-leaf nodes are kept sorted and linked into a singly linked-list via 8-byte `next` pointers. To support variable-sized values, existing key-value stores often use a separate persistent allocator [1, 10] to manage the actual key-value pairs, and only store a pointer in the persistent index structure. Similarly, in μTree, the list-leaf nodes are organized into *key-pointer* pairs, where the pointers point to the actual KV items. Note that μTree also supports placing the variable-sized values directly in the list layer by allocating the space of each list node through the persistent allocator.

All tree nodes, including inner nodes and tree-leaf nodes, are placed in DRAM, and can be reconstructed via the list layer. So, we only need to guarantee the crash consistency of the list layer. Inserting or deleting a node in the list

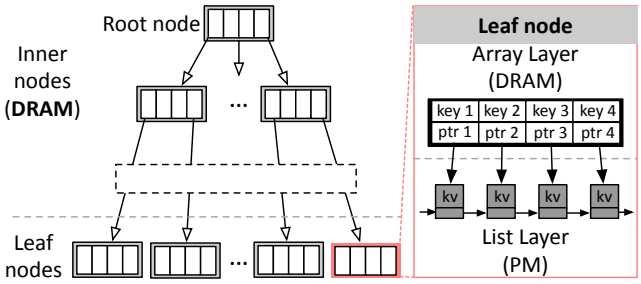


Figure 3: Overview of the μTree data structure.

layer involves modifying 8-byte `next` pointers, which are exactly the failure-atomic units. As a result, we do not need an extra logging mechanism to ensure the failure atomicity when updating the list layer. Several basic operations are described below (concurrency control is illustrated in §3.2):

- `Put` operation. Figure 4(a) shows the steps to insert a new key-value pair. It first finds the predecessor of the target node in the list layer by traversing through the B$^+$-tree (red dashed line in ❶). A new list-leaf node is then created and inserted into the list layer (i.e., ❷). In this step, we need to first flush the newly created list node out of the CPU cache, and then link it into the list layer by modifying and flushing the `next` pointer in its predecessor node, to ensure crash consistency. Finally, a new entry is inserted into the array layer, with the pointer pointed to the created list-leaf node (i.e., ❸). If this key has already been inserted by other concurrent threads, then this insertion is aborted and it restarts by updating the existing key-value pair. We can observe that we only need to issue two flushes to PM for each `Put` operation.
- `Get` operation. It first finds the entry in tree-leaf node with the given key, and then fetch the target list-leaf node with the pointer stored in the tree-leaf node. The `Get` operation can always read valid data because it executes in an opposite direction: In a `Put` operation, a created list node is made visible by inserting a new entry in the array layer on the final step; a `Get`, instead, reads the tree-leaf node before the list-leaf node.
- `Delete` operation. It is processed similarly as that of `Put` operations. Figure 4(b) shows the steps. The predecessor node is located in ❶, and we then delete the target list-leaf node by modifying the `next` pointer in its predecessor (in ❷). After this, the related entry in the array layer is deleted (in ❸).
- `Scan` operation. μTree processes range queries by traversing through the list layer directly. It first finds the starting entry at or after k provided by the query through the tree layer, and then probes via the `next` pointer in the list layer until reads n key-value pairs. Similar to Masstree [40]'s implementation, range scan operations in μTree are not atomic with respect to concurrent inserts and updates. For example, when a concurrent `Delete` operation has already deleted the list node but the corresponding entry in the tree-leaf node has not been deleted yet, such an inconsistent state is invisible to the current `Scan` operation and the range query will finally return n key-value pairs without this unfinished deleted items. μTree only supports forward range queries (backward range queries can be supported by adding backlinks in the list layer).

Finally, we analyze how μTree is suitable to deal with the tail latency problem. The key to improving the tail latency

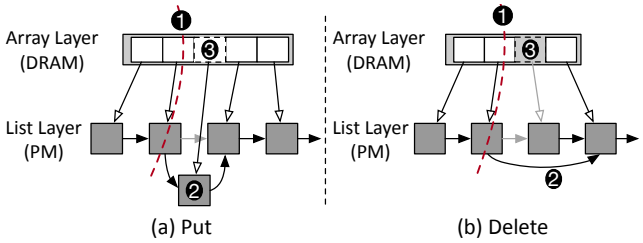


Figure 4: Process of Put/Delete operations.

is avoiding SROs in PM. By placing inner tree nodes and the array layer in DRAM, such overhead is mitigated, since DRAM has much higher write bandwidth and lower latency. Besides, B$^+$-tree delivers high cache locality by organizing tree nodes in an *array-based* way, so the overhead of chasing pointers to find a specific item is reduced. With such a design, however, the volatile array layer loses its ability to store data durably. Hence, we still need another index layer in PM to ensure durability. We choose linked-list for the following reasons: The index layer in PM should be simple enough, without causing extra SRO or consistency overhead. It does not need to provide fast query speed since the volatile B$^+$-tree is responsible for this. As a result, we use the singly linked-list that owns the *element-based* organization. Moreover, by traversing through the linked-list, the B$^+$-tree layer can be reconstructed when it is lost in power/system failures.

3.2 Coordinated Concurrency Control

The shadow list layer effectively tackles the problem of SRO overhead in PMs. However, it's still challenging to scale μTree to multi-cores while keeping its tail latency low. We cannot rely on existing locks inside a B$^+$-tree to coordinate concurrent accesses to μTree because 1) in a B$^+$-tree, the entries in the same leaf node typically share the same lock, and 2) the SROs block the execution of normal operations, which have been discussed in Section 2.3. To this end, we redesign the concurrency control protocol inside μTree, and propose the *coordinated concurrency control* mechanism. This is based on the following observations:

1) PM exhibits high write latency, especially with high-concurrency workloads. The list layer in PM, however, consists of independent list nodes, and are linked via 8-byte `next` pointers. Hence, it inherently supports fine-grained lock-free concurrency control using a sequence of `CAS` instructions to perform atomic updates.
2) DRAM, instead, exhibits much higher throughput and lower write latency, and the interference between concurrent threads is less severe. Hence, we can still reuse the existing coarse-grained lock to coordinate the volatile B$^+$-tree part.

In the following parts, we depict how coordinated concurrency control is designed to cope with write-write conflicts and read-write conflicts.

3.2.1 Write-Write Conflicts

A `Put` operation incurs two flush operations to the linked-list, which are the most time-consuming parts: one is the node persistence, which persists the newly created list node itself (including the `next` pointer that links it to the next list node). The other is persisting the `next` pointer of its predecessor node, which makes the newly created node accessible in the list. *Coordinated concurrency control* allows μTree

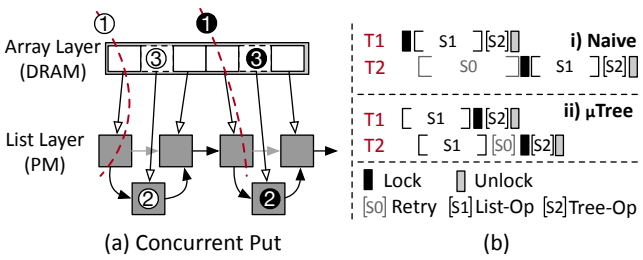


Figure 5: Write-Write conflicts (The right part compares the timing diagram of μTree and a naive approach.).

to flush data speculatively before acquiring locks. Thus, the persistence overhead is excluded from the critical write locking path, without interfering with other threads.

Figure 5(a) shows how two threads concurrently insert two different key-value pairs that belong to the same tree-leaf node. Among them, the first thread (denoted as T1) follows the steps of ①, ② and ③, and the second thread (denoted as T2) follows the steps of ❶, ❷ and ❸. We focus on the details of concurrency control in this part, despite that the steps are almost the same as that of Section 3.1.

In the first step (i.e., ❶/①), each thread optimistically finds the predecessor without acquiring any locks in the B$^+$-tree. This is achievable since many existing B$^+$-tree data structures support lock-free reads [23,40]. In the second step (i.e., ❷/②), they insert the newly created list-leaf nodes into the linked-list. We use `CAS` instruction to modify the `next` pointer in the predecessor nodes. In this way, the atomicity of insertion in the list layer is guaranteed. If a `CAS` fails, indicating that the predecessor node is obsolete (e.g., another list node is inserted between them), we need extra steps to deal with such anomaly (described later). In the final step (i.e., ❸/③), the two threads try to acquire the lock of the corresponding tree-leaf node, and then insert a new entry to point to the newly created list node.

Figure 5(b) shows the benefits of such design by comparing it with a naive approach, which relies on existing locks inside the B$^+$-tree layer for concurrency control. Specifically, the naive approach acquires the lock in the leaf node before it inserts the new list node and new entry (as in ❷ and ❸). As shown in the upper part, we can observe that T2 experiences extra delay (i.e., retry) to acquire the lock, and the two `Put` operations are serialized despite that they update different key-value pairs. As shown in the lower part, by coordinating the concurrency control between the two layers, insertion in the list layer (`List-Op` in the figure), which is the most time-consuming part, can be processed in parallel. As a result, two independent operations are less likely to interfere with each other and the tail latency can be improved accordingly.

However, the coordinated concurrency control mechanism with speculative persistence causes inconsistency between the two layers, and there're two anomalies to be dealt with: `CAS` failure. As described before, a `CAS` may fail when another node is inserted in the list layer before the current node (as shown in Figure 6(a)). Such a case happens since we get an obsolete predecessor in ❶, during which another new list node is inserted, but the tree-leaf node has not been inserted yet. To address such an issue, this thread is required to retry in the array layer to find the newest predecessor, and then to finish to rest steps. This is different from a DRAM-based

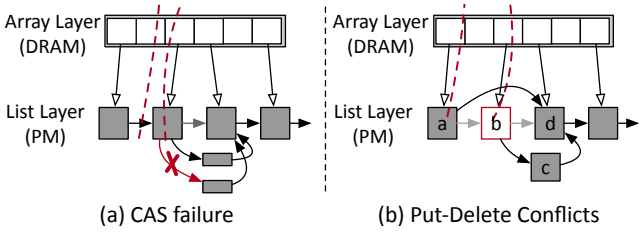


Figure 6: Two inconsistency scenarios.

lock-free linked-list, which typically traverses through the linked-list directly to locate the newest predecessor when a `CAS` fails. We do not allow this in PM because `CAS` cannot guarantee the 'persistent' atomicity, which is required in PM. Dirty read mistakes occur when a concurrent reader sees the result of an atomic modification before it is made persistent, and makes a persistent write based on this read. In μTree, only after a list node is inserted into the linked-list and made persistent, can it be inserted into the B$^+$-tree layer. Hence, *always retrying in the B$^+$-tree layer ensures that any visible nodes are persistent.* When concurrently inserting the same key-value pair, speculative persistence may incur extra tail latency due to the overhead of retrying, but such a case is rare.

`Put-Delete` conflicts is another well-known anomaly. As shown in Figure 6(b), one thread is trying to insert a list node (i.e., c). However, its predecessor (i.e., b) has been deleted by another thread. If we keep inserting this node, it will not be visible in the list layer. Such a case is not a big issue during the runtime, since other threads still have access to it through the B$^+$-tree layer. However, this node is lost if we rebuild the B$^+$-tree layer by scanning list nodes after a system failure. To address this problem, we follow a classical way [21] by adding a `Deleted` bit in the `next` pointer of each list node, and a thread modifies this bit before actually deleting it. Thus, other threads can identify the status of its predecessor and retry if it's deleted.

3.2.2 *Read-Write Conflicts*

Many existing B$^+$-tree structures support optimistic read, by encapsulating a `Version` field in each leaf node [23,40]. A `Put` operation sets the `Version` as dirty when updating the node, and cleans its state after the operation is completed. In PM, however, the time of a `Put` is unfortunately increased due to the high write latency of PM, which inevitably increases the abort ratio of `Get` operations.

Moving `Version` fields from tree-leaf nodes to list nodes supports more fine-grained optimistic reads. However, it causes higher space consumption. Regarding this, μTree proposes an *embedded version* mechanism leveraging the reserved bits in the `next` pointer of each list node. The layout of the `next` pointer is shown in Figure 7. Among them, the `Deleted` bit has already been illustrated in §3.2.1. The `Version` field contains 15 bits, which is increased both before and after the list node is updated by a concurrent thread.

When processing a `Get`, a reader acquires the pointer from the array layer and checks the embedded bits (i.e., `Version` and `Deleted`) both before and after reading data from the list node. The reader simply returns if the `Deleted` bit is set. If the `Version` field is odd, it will wait until the concurrent update is finished. The above steps ensure that any concurrent readers will see either the old or the new value, instead of an intermediate state. We can also observe that

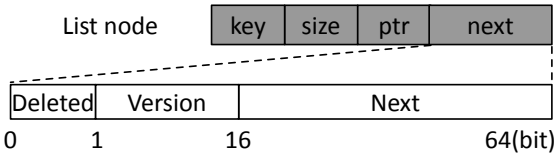


Figure 7: Layout of the `next` pointer in the list node.

by embedding such concurrency bits in the `next` pointer, concurrent reads and writes to different key-value pairs can be processed in parallel, even if they belong to the same leaf node in the B$^+$-tree layer.

3.3 Multi-Threaded Recovery

In μTree, only the persistent list layer survives after system crashes. To avoid the recovery overhead after normal shutdowns, μTree persists all the volatile tree nodes to a reserved position in PM. When the system restarts, the recovery thread directly copies it from PM to DRAM.

Recovery is unavoidable after unexpected system crashes. In this case, μTree has to scan all list nodes to rebuild the volatile nodes, similar to FPTree [43] and NV-Tree [57]. To reduce the recovery time, μTree adopts a *multi-thread recovery scheme*. It uses a set of persistent trackers to record the positions of some list nodes when processing `Put` operations. A list-leaf node is chosen to be a tracker after inserting a fixed number of key-value pairs (e.g., 0.1 million).

During the system recovery phase, recovery threads work in two phases. First, all valid trackers are sorted by the tracked keys, and are then dispatched to different threads for parallel recovery. Then, each thread iterates over the assigned list nodes, and reconstructs those disjoint partitions of the volatile B$^+$-tree, so as to rebuild the full B$^+$-tree.

4. IMPLEMENTATION

In this section, we describe how μTree is implemented. We discuss the base operations of μTree in §4.1, and give extra discussion in §4.2.

4.1 Base Operations

μTree can directly reuse the code of existing main memory B$^+$-tree as its volatile tree layer. Masstree, BwTree [38] and FAST&FAIR support lock-free read, which are the optimal choices. For a fair comparison, we directly place FAST&FAIR in DRAM to act as the volatile layer in μTree. We slightly modify it by adding a new interface of `optimistic_get()` (∼50 SLOC). It returns both the current list-leaf node and its predecessor with the given key. We do not need to ensure the strict consistency of this interface since we will recheck their adjacency through the `next` pointer in the list layer with `check_adjacency()` interface. We use `persist()` interface to flush data to PM by co-using `clwb` and `mfence` instructions.

`Put` (Update/Insert). Algorithm 1 describes the `Put` operation, which includes updating an existing item (Lines 5 - 9) or inserting a new item (Lines 10 - 20). To update an 8-byte value, μTree still relies on the `Version` field to prevent concurrent readers from reading data that hasn't been persisted yet, despite that an 8-byte pointer can be updated in-place via atomic instructions. For values larger than 8-byte, μTree cannot perform in-place updates since the updated value cannot be persisted atomically. Instead, μTree allocates a new list node to replace the original one, and this

Algorithm 1: Put(Key K, Value *V)

```
1  retry = 0;
2  RETRY:
3  (pre_node, cur_node) = btree→optimistic_get(K);
4  check_adjacency(pre_node, cur_node);
5  if cur_node != NULL then
       /* update an existing item.                          */
6      Acquire(cur_node→next→Version);  // Atomic add
        if Version is even, retry otherwise.
7      cur_node→ptr = V;      // 8-byte, in-place update.
8      persist(cur_node);
9      Release(cur_node→next→Version);       // Atomic
        add.
10 else
       /* insert a new item.                                */
11     if retry == 0 then
12         new_node = palloc();      // alloc a list node.
13         init(new_node, K, V);       // fill each field.
       /* [a,b,c] means [Deleted, Version and Next].        */
14     new_node→next = [0,0,pre_node→next→Next];
15     persist(new_node);                    // 1st flush.
       /* '*' means the original value.                    */
16     if !CAS(pre_node→next, [0,*,*],
        [0,*,new_node]) then
17         retry += 1;
18         goto RETRY;                // CAS failure, retry.
19     persist(pre_node→next);                // 2nd flush.
20     btree→insert(K, new_node);        // made visible.
```

is achieved by using similar steps as that of inserting a new item (pseudo-code is not shown separately). For brevity, some loose ends are not described in detail when inserting a new item. For instance, when a `CAS` fails (Line 16), it retries and finds that this item has already been inserted by others, so it needs to free the allocated space in Line 11.

`Get`. Algorithm 2 depicts the `Get` operation. Just like the existing optimistic read approach, it checks the statuses of the embedded bits before and after reading an item.

Algorithm 2: Get(Key K, Value *V)

```
1  (_, cur_node) = btree→optimistic_get(K);
2  if cur_node != NULL && cur_node→key == K
    then
3      RETRY:
4      oldVersion = cur_node→next→Version;
5      if (oldVersion % 2) != 0 then
6          goto RETRY;                  // updating, retry.
7      if cur_node→next→Deleted then
8          return NULL;                 // deleted, return.
9      copy(V, cur_node→ptr, cur_node→size);  // read.
10     if cur_node→next→Version != oldVersion then
11         goto RETRY;                  // modified, retry.
12     if cur_node→next→Deleted then
13         return NULL;                 // deleted, return.
14 return V;
```

`Delete`. Algorithm 3 shows how a `Delete` operation is processed. When deleting an item, a `CAS` instruction (Line 5 or 7) may fail when other concurrent threads are updating/deleting the `cur_node` or `pre_node`. For brevity, we just let the deleting thread to restart from the top. However, it's unnecessary to do so when a thread is updating this item. Instead, polling on the `Update` bit is enough.

Algorithm 3: Delete(Key K)

```
1  RETRY:
2  (pre_node, cur_node) = btree→optimistic_get(K);
3  check_adjacency(pre_node, cur_node);
4  if cur_node != NULL then
5      if !CAS(cur_node→next, [0,*,*], [1,*,*) then
6          goto RETRY;
7      if !CAS(pre_node→next, [0,*,*],
        [0,*,cur_node→next→Next]) then
8          cur_node→next→Deleted = 0;    // reset bit.
9          goto RETRY;                   // CAS fail, retry.
10     persist(pre_node→next);
11     btree→del(K); free(cur_node);
```

4.2 Discussion

Memory allocation consistency. There are two inconsistency cases of the memory space in μTree, including the *memory leak* and *read after delete*. *Memory leak* means that the allocated memory space fails to be freed properly due to system crashes. For example, an insertion may fail when a new node has been made persistent but not been linked to the list yet. The newly allocated node is lost after rebooting. One naive way to solve this issue is to persist the allocation metadata whenever a (de)allocation occurs, but with expensive persistence cost. μTree proposes a chunk-based allocation strategy, which acquires a large chunk of persistent memory from the persistent allocator (e.g., PMDK [24]) when existing memory chunks use up. The metadata (address and size) of such chunks is organized in a persistent linked-list. These chunks are further cut into variable-sized blocks, which are organized into a volatile free list to serve allocation/free requests of the list layer. During the system recovery, both the chunk metadata and the list layer in μTree are scanned to distinguish the used and freed memory blocks. The *read after delete* issue is the case that physically deleted nodes are read by other concurrent threads. μTree borrows the idea of an epoch-based space reclamation [20], which reclaims the deleted nodes in different phases, thus ensuring the safety of reclamation on garbage nodes.

The correctness of concurrency control. In μTree, the volatile tree layer reuses the code of FAST&FAIR, which ensures the correctness of the concurrency control in DRAM. The lock-free linked-list has also been implemented many times in the past. Hence, we focus on the correctness of the coordination between the two layers. For search operations, μTree guarantees that any visible list nodes from the array layer are in their newest states and have been persisted to PM. Reads are blocked if there are other concurrent writers, by using the embedded access control bits. For concurrent writes, coordinated concurrency control with speculative persistence may cause inconsistency between the array and list layer. However, when an operation aborts in the list layer (i.e., `CAS` failure or the embedded bit has been set), we always retry from the array layer. Hence, the two

layers will be consistent eventually. To sum up, the correctness of μTree is guaranteed so long as the concurrency control in the tree and list layer is correct.

μTree's semantics. A `Put` or `Delete` operation is considered as committed once the corresponding list nodes in PM have been updated. If a system failure occurs before the volatile tree nodes in DRAM are updated, the tree layer still can be recovered to its newest state by scanning the list layer, which contains all the committed nodes. With the decoupled design in μTree, however, the committed list nodes are visible only after the tree layer has been updated. Note that this does not violate the linearlizability property, which requires that the competed updates should be visible by subsequent readers (the *complete point* does not necessarily to be the same as the *commit point*).

Extra space consumption. Similar to the past work [22, 43, 55], μTree adopts a hybrid architecture that places index nodes in both DRAM and PM. As a result, μTree consumes extra DRAM space. Apart from the performance benefits gained from μTree's design, we argue that such space overhead is still acceptable for the following reason: Production workloads (e.g., Facebook ETC Pool [42]) typically have a wide distribution of value sizes ranging from several bytes to multiple megabytes. In these variable-sized values, large values typically dominate in terms of space consumption. As a result, the index itself only occupies a small portion of the space compared to the actual key-value pairs. We will evaluate the space overhead in §5.4.

The idea of using a *shadow index layer* has been adopted many times in the past [22, 55]. They duplicate the index either for improving the indexing efficiency [55], or for better read performance [22]. μTree is novel in improving the tail latency by coordinating the tree and list layer.

5. EVALUATION

In this section, we evaluate the overall performance of μTree against the state-of-the-art persistent indices in Section 5.2 and Section 5.3. We also evaluate the memory space efficiency (§5.4), recovery overhead (§5.5) and μTree's performance in a real-world application (§5.6).

5.1 Experimental Setup

Testbed. We ran our experiments on a machine with the actual persistent memory — Intel Optane DC Persistent Memory [41] — and the second-generation Xeon Scalable processors. Optane DCPMMs are configured in 100% App Direct mode [26], so that software has direct byte-addressable access to PM. Table 1 reports the relevant details of our test machine. The six Optane DCPMM modules are evenly attached to two CPU sockets (i.e., each CPU socket owns three modules), and their spaces are managed into two regions (i.e., `pmem0` and `pmem1`). In our evaluation, applications access PM by first using a PM-aware file system (e.g., Ext4-DAX in this paper) to manage the pmem device and then relying on PMDK [1] to allocate the PM space, which provides basic interfaces (e.g., `pmemobj_alloc()`) to applications. We only allow client threads to allocate PM space from their local pmem device, since recent work reveals that cross-NUMA accessing to Optanes impacts overall performance dramatically [56].

Compared Systems. In our tests, we compare μTree with state-of-the-art persistent indices, including wB$^+$-Tree [51], NV-Tree [57], FAST&FAIR [23], and FPTree [43]. Among them, FAST&FAIR is open-sourced[1] and we directly use their code. Single-thread version of FPTree, wB$^+$-Tree and NV-Tree have been implemented by Liu et al. [39] and we borrow their code[2] directly. We also implement a concurrent version of FPTree based on STX B$^+$-Tree[3] and incorporates the key design decisions in their paper. The performance of a persistent skip list [46] is also given in this part. By default, we use 8-byte keys. The default number of keys in each tree node is set to 64. We only store 8-byte pointers as values in the evaluated persistent indices, which can be used to point to actual key-value pairs.

Table 1: Description of the evaluation platform.

		Description
CPU	Type	2× Intel Xeon Gold 6240M
	# of physical cores	36 (hyper-threading disabled)
	Frequency	3.3 GHz
	Caches	L1: 32KB Icache, 32KB Dcache L2: 1MB, L3:25MB (shared)
MEM	PM Capacity	1.5 TB (6 modules)
	PM Read Latency	302 ns (Rnd/8-byte/1 thread)
	PM Write Bw	13.2 GB/s (4KB sequential)
	DRAM Capacity	192GB (32 GB/DIMM)
OS	Release Version	Ubuntu 18.04.3 LTS
	Kernel	Linux 4.15.0

Tail Latency Measurement. Measuring tail latency often adds non-negligible overhead to the tested operations. If we simply record start and end timestamps of each operation, the measured throughput is typically lower than its actual result, which again impacts the latency distribution, since many designs trade latency for throughput. As a result, we take a sampling approach that samples 10% of the requests uniform randomly, as done in [37].

5.2 Tail Latency and Throughput Analysis

Overall Evaluation. We first evaluate the latency distribution of each persistent index with a workload of zipfian key distribution (parameter = 0.99). As we target at write-intensive workloads, we only evaluate the `Put` operation in this part. We increase the number of threads from 1, 4, 8, 12, ..., to 36 with a step of 4. The median, 90th, and 99th percentile latencies, as well as the corresponding throughput of the evaluated indices are collected. Figure 8 shows the results and we make the following observations:

1) μTree exhibits much lower tail latency and significantly higher throughput. When running at a target load of 2.2 Mops/s, μTree's 99.9th percentile latency is only 5 μs, which is one order of magnitude lower latency than that of FPTree. Put differently, for an SLO on the 99th percentile latency of 30 μs, μTree can perform 7 Mops/s, 2.8× and 4.7× faster than FAST&FAIR and FPTree, respectively. μTree achieves the best performance by overcoming the limitations of existing designs when dealing with the overhead of SROs and inter-thread interferences, which have been discussed in Section 2.3. What's more, μTree achieves a peak throughput of 7 Mops/s, while the throughput of FAST&FAIR and FPTree peaks only at 3.1 Mops/s and 2.2 Mops/s, respectively. μTree

[1] `https://github.com/DICL/FAST_FAIR`
[2] `https://github.com/liumx10/ICPP-RNTree`
[3] `https://panthema.net/2007/stx-btree/`

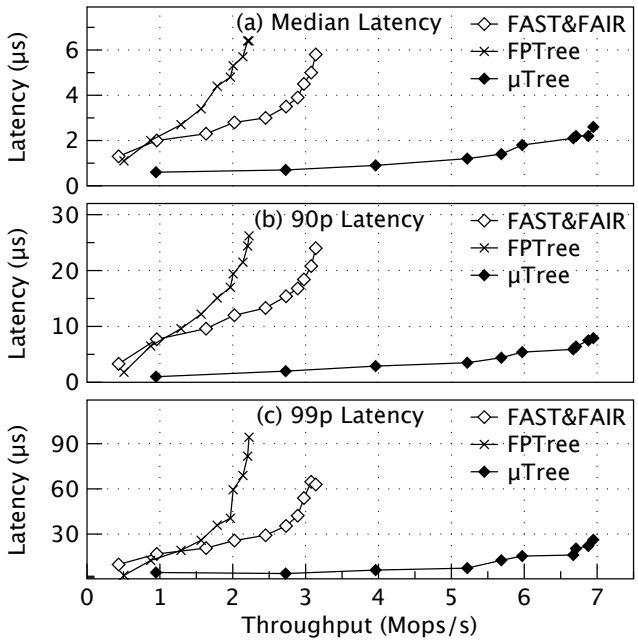


Figure 8: Throughput vs. median/ 90th/99th percentile latency of `Put` operations for different persistent indices.

achieves higher peak throughput because it places tree nodes completely in DRAM. The list layer in PM does not incur SRO overhead and supports more fine-grained concurrency control, and thus delivers higher scalability. Note that FAST&FAIR exhibits higher peak throughput than FPTree despite that it places all data in PM. FAST&FAIR avoids logging and read locks completely, FPTree, instead, needs to perform expensive logging when leaf nodes split. The results here are also consistent with that of the original FAST&FAIR paper.

2) All the evaluated persistent indices show a wide spectrum of latency distribution. For FAST&FAIR, the 90th and 99th percentile latencies are averagely 4× and 10× higher than its median latencies, respectively. μTree also exhibits much higher tail latency, despite that the absolute figures are much smaller than the other two indices. This is because the SROs still incur higher execution time, even if they are performed in DRAM.
3) The tail latency problem is aggravated at load with higher concurrency. With one thread, the 99th percentile latencies of FAST&FAIR and FPTree are 7.5× and 2.4× higher than their median latencies, respectively. However, such factors improve to 10.1× and 9.1× with 36 threads. This reflects that the inter-thread interferences are the main culprit leading to the high tail latency. Such a ratio only increases slightly for μTree, since it incorporates the coordinated concurrency control to reduces the interference overhead.

Sensitivity to Update Ratio. We then analyze how these persistent indices behave in terms of 99th percentile latencies when the Put/Get ratio varies. Similarly, we use the workload with the zipfian key distribution (parameter = 0.99). The results are shown in Figure 9 (Put:Get = 100:0 is shown in Figure 8(c)). We observe that all the evaluated indices exhibit higher peak throughput and lower tail latency as the percentage of Get increases. This is because PM has much higher read bandwidth and `Get` operation does

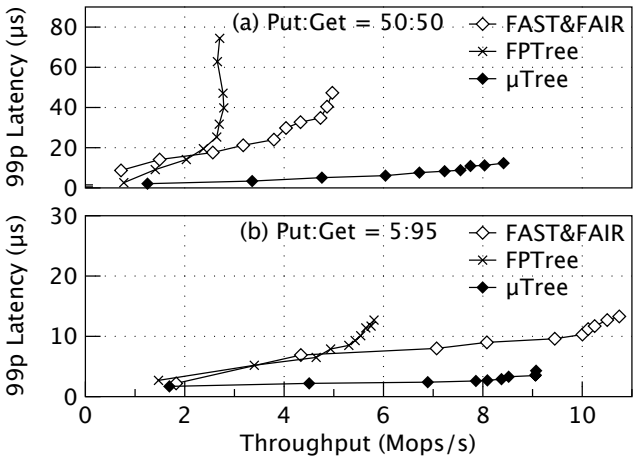


Figure 9: Throughput vs. 99th percentile latency of `Put` operations with varying Put/Get ratio.

not cause SRO overhead. In addition to the above intuitive findings, we also notice that 1) Compared to FPTree, the peak throughput of FAST&FAIR is more sensitive to the Get ratio. Specifically, it outperforms FPTree by 40% with 0%-Get workload (Figure 8(c)), but achieves 1.75× higher peak throughput with 95%-Get workload. FAST&FAIR incorporates the FAST mechanism, which enables lock-free reads, so it's more efficient in handling Get operations. FPTree, instead, relies on `mutex` locks to coordinate concurrent Get operations. 2) With 95%-Get workload running at 9 Mops/s, the 99th percentile latency of μTree is 2.7× lower than FAST&FAIR. We observe that their tail latencies are mainly decided by Put operations: FAST&FAIR exhibits higher latency for Put operations since it places the whole tree in PM. In terms of throughput, we notice that FAST&FAIR even delivers higher peak throughput than μTree, this is because μTree introduces an extra memory access from the tree-leaf node to the list node for each Get operation.

Sensitivity to Skewness. Third, we evaluate μTree by changing the hotness of key distribution. Figure 10 reports their performance with both zipfian key distribution (parameter = 0.9) and uniform workloads. Parameter = 0.99 is shown in Figure 8(c). We observe that:

1) FAST&FAIR exhibits almost unchanged peak throughput despite the key distribution, since its throughput is mainly restricted by the PM write bandwidth (we will analyze the sensitivity to PM bandwidth at the end of this part).
2) The 99th percentile latencies of FAST&FAIR and FPTree decrease as the key distribution shifts from zipfian to uniform. This is easy to understand: with the skewed workload, the chance of conflicts between threads is higher. Thus, a thread is more likely to be blocked by other concurrent threads and causes higher tail latency.
3) μTree achieves higher peak throughput under the uniform workload, since it causes less PM writes, and the PM bandwidth is still under-utilized. μTree also delivers lower tail latency compared to the other two indices.

Sensitivity to PM Bandwidth. Finally, we analyze how these indices behave with varying PM bandwidth. We change the PM bandwidth by adding a different number of Optane modules. Figure 11 shows the results with 1 and 2 Optane modules, respectively (each module achieves a write

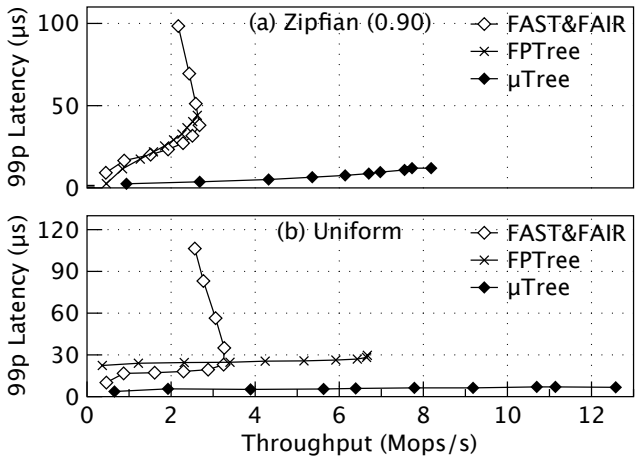


Figure 10: Throughput vs. 99th percentile latency of `Put` operations with varying key distribution.

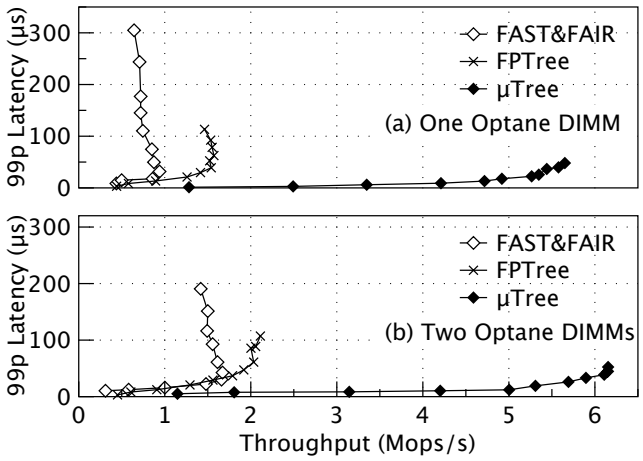


Figure 11: Throughput vs. 99th percentile latency by varying the number of Optane DC DIMMs (Put operation and parameter = 0.99).

bandwidth of 2.2 GB/s). By comparing with the results in Figure 8(c) (six Optane modules), we observe that the peak throughput of FAST&FAIR decreases by 67%, while μTree drops only by 18% as the number of Optane DIMMs reduces from 6 to 1. In terms of tail latency, both μTree and FPTree exhibits almost unchanged 99p latencies, while FAST&FAIR shows more than 300μs of tail latency when we use only one Optane module. As a whole, with the optimizations in μTree, its performance is more robust as we change the PM hardware bandwidth.

5.3 Single Thread Evaluation

Figure 12 reports the performance of each index using a single thread when running individual operations (i.e., 100% insert, update, delete, search or scan). A persistent skip list [46] is also compared in this part. In the skip list, only the lowest-level list nodes are placed in PM, and we only ensure the consistency of the lowest list. The skip probability is set to 50%. We first insert 10 million key-values, and then perform 10 million individual operations. We also use Intel's PCM tool [25] to measure the number of accesses to PM to help with understanding the performance behavior (see the table in Figure 12, we only give the results of insert and search operations, due to the limited space).

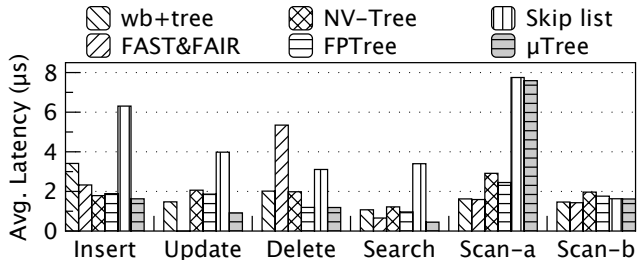
• `Insert`. Insert latency is directly affected by the number of flushes to PM [37]. As expected, wB$^+$Tree and FAST&FAIR exhibit higher insert latency since they place all tree nodes in PM and thus incurs much more PM writes (see the table). FPTree and NV-Tree, instead, only place leaf nodes in PM, and thus show lower latency. Specifically, NV-Tree inserts an entry by first appending it at the end of the leaf node, and then updating the size of the node, with a total of two flushes. FPTree requires three flushes per insert in normal cases (entry, bitmap, and fingerprint), so it shows a higher latency than NV-Tree. μTree achieves the lowest insert latency among the compared systems. First, μTree only requires two flushes to insert an entry, which is similar to that of NV-Tree. Second, μTree adopts the list format in PM to eliminate sort and balance overhead. Skip list has the highest latency since it uses linked-lists at different levels, which incurs expensive pointer chasing overhead (26.5 PM reads per insertion).

• `Update`. Compared to inserts, an update only operates on an existing key. wB$^+$Tree's update latency is lower than that of inserts since each update only requires 3 flushes (validity bit does not need to be flushed). NV-Tree performs updates slower than inserts because it uses a combination of deletion and insertion to handle updates. FAST&FAIR and μTree assume values stored in an index are 8-byte pointers, so each update is handled via an atomic operation directly, which only requires one flush. As a result, μTree achieves the lowest update latency among the compared indices. FAST&FAIR does not implement this operation so we do not report its result. Actually, all existing indices can use atomic instructions as an alternative optimization to handle updates with 8-byte values.

• `Delete`. Again, μTree achieves the lowest delete latency. A deletion in μTree simply modifies and persists a predecessor pointer atomically, while other index structures adopt more complicated consistency-aware deletion strategies: NV-Tree appends a tombstone at the end of the node to indicate a deletion with two flushes; FAST&FAIR needs to shift the entries to keep the nodes ordered. wB$^+$Tree shows almost the same latency as that of update operation, since it simply updates the indirection slotted array and the bitmap to mark the corresponding slot as deleted. We also notice that FPTree achieves lower latency than other indexing structures, since it only needs to reset the bitmap when performing deletion operations. It also avoids merge operations to reduce the persistence cost. However, this compromise may lead to low space utilization of tree nodes under some deletion-intensive workloads (in §5.4).

• `Search`. μTree shows lower search latency than wb$^+$-Tree, FAST&FAIR, NV-Tree and FPTree, and gains noticeable performance improvement over the skip list. This is as expected: Each search operation in a skip list causes 26 PM reads on average, while PM has much higher read latency (Optane DCPMM's read latency is more than 300 ns). Other compared B$^+$-tree indices also incur more PM reads than μTree. For example, FAST&FAIR and wB$^+$Tree place all tree nodes in PM; FPTree and NV-Tree keep entries in each leaf node unsorted, so they need to probe linearly to find a specific entry, which needs to load more data from PM and waste extra CPU cycles. Note that FAST&FAIR incurs comparable PM reads as that of wB+Tree but shows much lower search latency, because it uses lock-free reads.

• `Scan`. Range scans start at a random initial key and read



☞		wb+tree	F&F	NV-Tree	FPTree	Skiplist	μTree
I	PM Writes	5.9	8.5	3.4	5.2	3.4	2.1
	PM Reads	21.2	15.6	4.9	10.9	26.5	3.2
S	PM Reads	14.0	15.6	22.6	8.3	26.6	0.9

☞ The number of PM cache line reads/writes for each search and insert operation by using Intel PCM tool [25]. I: Insert, S: Search.

Figure 12: Single thread performance of Insert/Update/Search/Delete/Scan operations.

the following 20 records. We test two cases: In `Scan-a`, key-value pairs are inserted randomly before testing scan operations, In `Scan-b`, key-value pairs are inserted in sequential order. wB$^+$Tree and FAST&FAIR return records in sorted order directly by keeping entries in tree nodes ordered (wB$^+$Tree uses indirection slotted arrays). As a result, they show lower scan latency than FPTree and NV-Tree, which must perform an additional sorting step before returning the results. Both skip list and μTree are sensitive to the initial order of inserted key-value pairs. When the items are inserted in sequential order (in `Scan-b`), both of them show comparable latency to other indices, since adjacent list nodes are stored in contiguous memory space, still benefiting from the spatial locality. However, if the items are inserted randomly (in `Scan-a`), skip list and μTree need to walk through the list layer to get all the records, which causes a random PM read for each key-value pair, and thus deliver the highest latency. Note that μTree still can overcome such a problem by duplicating 8-byte values in the volatile B$^+$-tree layer to exploit the cache locality.

5.4 Space Efficiency

Memory Consumption of DRAM and PM. Figure 13 shows the DRAM and PM consumption of different index structures. In this part, we measure the space consumption by including actual key-value pairs as well, which reveals the case of real-world deployments. Since over 90% of KVs in production workloads are less than 500B in Facebook [7], the value size is set from 64B to 1KB.

From Figure 13, we make two observations. (1) The list-based management in PM is not worse than the tree-based organization regarding space consumption. After performing one million inserts with 64-byte value, μTree uses 83.3 MB of PM space, while wB$^+$-Tree, FAST&FAIR, NV-Tree, FPTree and skip list consumes 90.7, 84.5, 94.5, 84.9, 83.3 MB of PM space, respectively. Due to the balance constraint in B$^+$-tree, there is a large portion of memory space wasted in the tree nodes. (2) The space consumption of persistent memory is the dominant part. The DRAM consumption remains almost constant with varied value sizes, while the PM consumption increases as the value size grows. This is because values have larger sizes than keys and pointers. Even for μTree, which keeps the whole tree layer in DRAM, the DRAM consumption is relatively small compared to

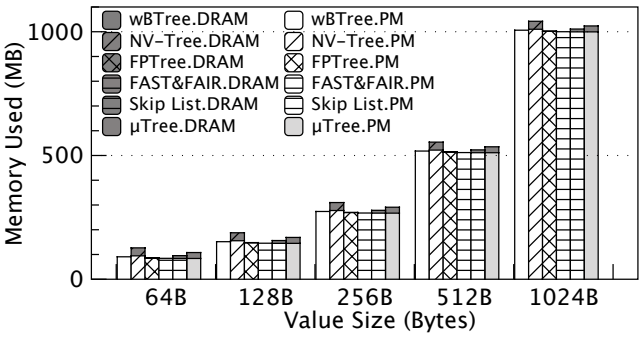


Figure 13: Space consumption of DRAM and PM.

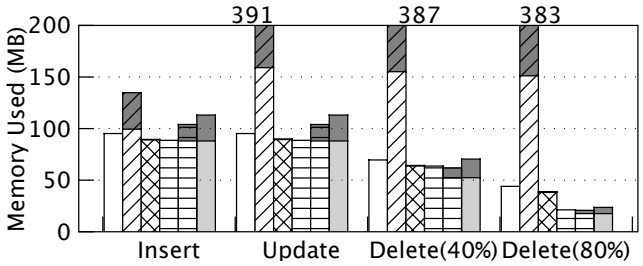


Figure 14: Memory utilization after deletions

PM. After performing one million inserts with 64/128/256/512/1024-byte values, the DRAM consumption occupies 22.3%/14.3%/8.2%/4.5%/2.3% of the total space.

Memory Utilization after Deletions. Figure 14 shows the memory space utilization of different indices in four stages: the Insert stage (after inserting 1 million entries), the Update stage (after updating 1 million entries), the Delete (40%) stage (after deleting 40% entries), and the Delete (80%) stage (after deleting 80% entries).

From Figure 14, we observe that the list-based organization, such as in μTree and skip list, has higher memory utilization than a tree-based organization, especially after long runs with delete operations. This is because persistent B$^+$-Trees use consistency techniques to prevent space reclamation. Specifically, NV-Tree uses an append-only strategy for crash consistency, which does not free the deleted entries immediately. wB$^+$-Tree and FPTree do not perform merge operation when the number of keys in a node is below a minimum threshold (e.g., 50% in standard B$^+$-Tree), so as to avoid the extra flush overhead. However, this compromise leads to higher space consumption after 40% or 80% deletions. FAST&FAIR aggressively merges operations to achieve high space utilization, but it leads to poor deletion performance (in Figure 12).

5.5 Recovery Overhead

Figure 15 shows the rebuild time of μTree with a different number of KV pairs and recovery threads. Note that the recovery process only needs to rebuild the volatile tree layer, which does not incur any PM write. The recovery process only needs 0.06/0.48/4.2 seconds to rebuild a tree containing 1/10/100 million KV pairs with 24 threads, respectively. Most of the recovery time is spent on chasing the sibling pointers in each list node to scan the whole list layer. The multiple-thread recovery scheme efficiently reduces the scan cost to speed up the recovery process. μTree may deliver intolerable recovery time under extreme workload (e.g., all the KV items are extremely small). However, production workload (e.g., Facebook `etc` pool [7]) typically

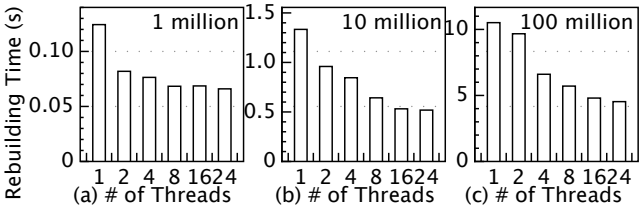


Figure 15: Recovery overhead analysis.

have a wide distribution of value sizes (small items dominate in terms of number, while large ones dominate in terms of space consumption), and thus has less index to recover than the aforementioned extreme case.

5.6 The Key-Value Store Evaluation

We use Redis [49], an in-memory key-value store to evaluate μTree in real-world environments. Redis is modified to support multi-thread execution, and we replace its storage engine with our index structures. We choose the Session-Store workload (update-heavy, 50:50 read:update) of YCSB benchmark [13] for evaluation. The YCSB clients and the Redis server are deployed on the same server to eliminate the network affects. Figure 16 shows the throughput and the corresponding 99th percentile latency of Redis with different persistent index structures. μTree achieves the highest peak throughput (670 Kops/s) and the lowest tail latency. Specifically, for a target load running at 500 Kops/s, μTree restricts its 99th percentile latency within 50 μs, 50% and 22% lower than that of FAST&FAIR and FPTree. Put differently, for an SLO on the 99th percentile latency of 60 μs, μTree can perform 670 Kops/s, 1.86 times and 1.26 times the throughput of FAST&FAIR and FPTree.

6. RELATED WORK

Tail Latency Optimizations. Many recent work proposed different approaches in the operating system to optimize tail latency [27, 44, 45, 47]. In key-value stores, the value sizes also affect tail latency. Minos [17] improves tail latency by using dedicated cores to process large values.

Persistent Indexing Structures. CDDS-Tree [53] is the first consistent and persistent B$^+$-Tree in persistent memory that associates each entry with a version number. However, versioning incurs high write traffic. Thus, a number of research efforts are proposed to reduce PM writes, including NV-Tree [57], wB$^+$-Tree [51], FPTree [43], FAST&FAIR [23], bzTree [6], and RNTree [39]. DPTree [59] is a new persistent index that adopts batching to reduce the number of expensive PM writes required for crash consistency. The core idea behind DPTree is a two-level persistent index: writes are first batched in an in-DRAM buffer tree, and are later merged into a base tree in PM. DPTree is well optimized for throughput, however, the background merging process may stall front-end operations and consumes extra PM bandwidth, which instead impacts tail latencies of front-end operations. RECIPE [35] presents an approach to convert concurrent DRAM indices into crash-consistent ones for PM.

Several recent systems duplicate indexing items in two data structures to deliver high performance or improve functionality. HiKV [55] consists of a B$^+$-Tree index in DRAM and a hash index in PM. It leverages the hash index to hide high write overhead of persistent B$^+$-Tree, and relies on the B$^+$-Tree to support scan operations. Before performing a scan, it needs to block all updates to the hash index until all the updates made before have been synchronized to the B$^+$-Tree. Bullet [22] tries to improve the read performance of a KV store by managing two hash tables simultaneously (one in PM and the other in DRAM), and using cross-referencing logs (CRLs) to keep the two hash table consistent. PmemKV [14] also shadows the keys of leaf nodes in DRAM. Similar to these approaches, μTree also duplicates indices, but with the design goal of improving tail latency.

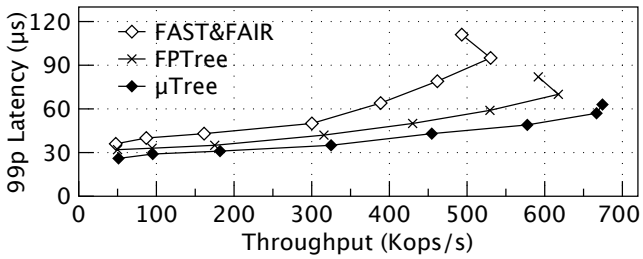


Figure 16: Throughput vs. 99th percentile latency with Sessionstore workload.

Concurrency Control. In addition to the naive concurrency control in B$^+$-Trees that locks the whole tree, recent research, such as B^{link}-Tree [36], OLFIT [11] and Masstree [40], propose version-based optimistic read and per-node write locking. Besides, FPTree leverages HTM to handle concurrent accesses to inner nodes, and per-node locks to serialize concurrent updates to the leaf nodes.

A number of research efforts are proposed to design lock-free B$^+$-Tree, but they lead to either performance or load imbalance problems. PALM [50] is a lock-free concurrent B$^+$-Tree based on the Bulk Synchronous Parallel model, which leads to higher query latency than that of Masstree. Both Cassandra [32] and Megastore [8] use keyhash-based approach to partition the B$^+$-Tree index among multiple cores. However, such a design cannot support scan operations effectively. SLIK [28] uses range partition on ordered multiple B$^+$-Tree indices, which will incur load imbalance problems. List-based indexing structure, such as skip list [46], supports lock-free concurrency control using atomic instructions.

7. CONCLUSION

We observe that persistent B$^+$-tree suffers long tail latency and we specify two root causes: 1) the internal structural refinement operations (SROs) and 2) the inter-thread interferences. In this paper, we propose μTree to address such problems. First, a shadow linked-list layer is introduced to the leaf nodes of a B$^+$-tree to minimize the SRO overhead. Second, we propose a coordinated concurrency control mechanism to reduce the interference between threads. Our evaluation shows that μTree significantly lowers the 99p tail latency, and achieves 2.8 - 4.7× higher throughput than the state-of-the-art persistent indices. Source code is available at `https://github.com/thustorage/nvm-datastructure`.

Acknowledgment

We sincerely thank the anonymous reviewers for their feedback and suggestions. This work is supported by National Key Research and Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61772300, 61832011), Huawei, and Special Topics of Major Scientific and Technological Projects in Sichuan Province (Grant No. 2018GZDZX0049).

8. REFERENCES

[1] The persistent memory development kit. "pmem.io".

[2] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 123–132, New York, NY, USA, 2012. ACM.

[3] H. Akinaga and H. Shima. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE*, 98(12), 2010.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). pages 63–74, 2010.

[5] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst.*, 9(2):13:1–13:35, May 2013.

[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] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[8] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc Conf Innovative Data Syst Res (CIDR)*, volume 11, pages 223–234, 01 2011.

[9] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter. Robinhood: Tail latency aware caching – dynamic reallocation from cache-rich to cache-poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, Oct. 2018. USENIX Association.

[10] 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 '16, pages 677–694, 2016.

[11] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. VLDB '01, pages 181–190, 2001.

[12] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *PVLDB*, 8(7):786–797, Feb. 2015.

[13] 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*, SoCC '10, pages 143–154, 2010.

[14] I. Corporation. Key/value datastore for persistent memory. https://github.com/pmem/pmemkv, 2020.

[15] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[17] D. Didona and W. Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, Boston, MA, Feb. 2019. USENIX Association.

[18] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, pages 401–414, 2014.

[19] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.

[20] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[21] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

[22] Y. Huang, M. Pavlovic, V. Marathe, M. Seltzer, T. Harris, and S. Byan. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 967–979, Boston, MA, July 2018. USENIX Association.

[23] 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, Oakland, CA, 2018. USENIX Association.

[24] Intel. The NVM Library. http://pmem.io/, 2016.

[25] Intel. Processor counter monitor (pcm). https://github.com/opcm/pcm, 2020.

[26] 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.

[27] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for usecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, Feb. 2019. USENIX Association.

[28] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. Slik: Scalable low-latency indexes for a key-value store. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 57–70, Berkeley, CA, USA, 2016. USENIX Association.

[29] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST '14, pages 33–45, 2014.

[30] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 427–444, Berkeley, CA, USA, 2018. USENIX Association.

[31] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceeding of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '13, pages 256–267, Apr. 2013.

[32] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[33] B. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30:131–141, Jan. 2010.

[34] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, 2009.

[35] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 462–477, New York, NY, USA, 2019. ACM.

[36] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, Dec. 1981.

[37] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm. Evaluating persistent memory range indexes. *PVLDB*, 13(4):574–587, Dec. 2019.

[38] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering*, ICDE '13, pages 302–313, 2013.

[39] 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*, pages 1–10, 2019.

[40] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, 2012.

[41] I. Newsroom. Intel® optane™ dc persistent memory. `https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html`, Apr. 2019.

[42] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 385–398, 2013.

[43] 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, 2016.

[44] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[45] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. ACM.

[46] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[47] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. K. Ousterhout. Arachne: Core-aware thread management. In *OSDI*, 2018.

[48] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, 2009.

[49] S. SANFILIPPO and P. NOORDHUIS. Redis. `http://redis.io`, 2009.

[50] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. 4:795–806, 08 2011.

[51] P. B. G. Shimin Chen and S. Nath. Rethinking database algorithms for phase change memory. In *Fifth Biennial Conference on Innovative Data Systems Research*, CIDR '11, pages 21–31, January 2011.

[52] 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, New York, NY, USA, 2013. ACM.

[53] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST '11, pages 61–75, 2011.

[54] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar. Tao: How facebook serves the social graph. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 791–792, New York, NY, USA, 2012. ACM.

[55] F. Xia, D. Jiang, J. Xiong, N. Sun, and T. Moscibroda. Hikv: A hybrid index key-value store

for dram-nvm memory systems. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '17, 2017.

[56] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, Feb. 2020. USENIX Association.

[57] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. 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 '15, pages 167–181, 2015.

[58] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 14–23, 2009.

[59] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen. Dptree: differential indexing for persistent memory. *PVLDB*, 13(4):421–434, 2019.