DPTree: Differential Indexing for Persistent Memory

Xinjing Zhou
Zhejiang University
zhouxj1@zju.edu.cn

Lidan Shou
Zhejiang University
should@zju.edu.cn

Ke Chen*
Zhejiang University
chenk@zju.edu.cn

Wei Hu
Alibaba Group
droopy.hw@alibaba-inc.com

Gang Chen
Zhejiang University
cg@zju.edu.cn

ABSTRACT

The emergence of persistent memory (PM) spurs on redesigns of database system components to gain full exploitation of the persistence and speed of the hardware. One crucial component studied by researchers is persistent indices. However, such studies to date are unsatisfactory in terms of the number of expensive PM writes required for crash-consistency. In this paper, we propose a new persistent index called DPTree (Differential Persistent Tree) to address this. DPTree's core idea is to batch multiple writes in DRAM persistently and later merge them into a PM component to amortize persistence overhead. DPTree includes several techniques and algorithms to achieve crash-consistency, reduce PM writes significantly, and maintain excellent read performance. To embrace multi-core processors, we present the design of concurrent DPTree. Our experiments on Intel's Optane DIMMs show that DPTree reduces PM writes by a factor of 1.7x-3x compare to state-of-the-art counterparts. Besides, DPTree has a competitive or better read performance and scales well in multi-core environment.

PVLDB Reference Format:
Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, Gang Chen. DP-Tree: Differential Indexing for Persistent Memory. *PVLDB*, 13(4): 421-434, 2019.
DOI: https://doi.org/10.14778/3372716.3372717

1. INTRODUCTION

The past decade has witnessed innovative memory technologies which led to the production and commercialization of *persistent memory* (PM) [4], to name a few, Phase Change Memory [29], Spin-Transfer Torque MRAM [5], and 3D XPoint [1] etc. Persistent memory features not only the durability seen in storage devices, but also byte-addressability, high density, and close-to-DRAM latency, thus enabling a wide range of new applications.

As PM could be larger than DRAM in capacity, one exciting application is to persist components of an in-memory database for capacity expansion, and recovery cost reduction [3, 2]. In particular, researchers study the problem of persistent index structures. Such indices provide all operations supported by main-memory indices, such as insert, lookup, and scan, on database tuples or key-value payloads. Also, they can recover from system failures much faster due to the persistence of the memory. Unfortunately, it is non-trivial to design an efficient and persistent index. As modern CPUs usually employ write-back cache for better performance, sudden system failures might cause inconsistency in the data structure in PM if writes did not reach PM in the correct order and in time. To guard against such inconsistency, existing PM index structures use *persist* primitive (e.g., `clwb+mfence` in x86) to carefully order their writes and synchronously flush cache lines to PM. However, such synchronous writes are shown to incur high-cost [9, 18] in latency and device endurance. Therefore existing studies strive to minimize the usage of these primitives.

Based on the structural property, existing studies on PM indices can be roughly classified as *B^+-Tree based*, *Trie-based*, and *hybrid*. Generally, B^+-Tree-based structures (e.g. CDDS-Tree [26], wB^+-Tree [9], NVTree [31], FPTree [23] and FASTFAIR [12]) are sub-optimal in terms of the number of *persist* primitives on the critical code path of index update. Trie-based structures (e.g., WORT [15]) achieve excellent write performance for certain key distributions at the expense of sub-optimal range scan efficiency. As for hybrid structures, the idea is to combine multiple physical index structures to form a logical index. For example, HiKV[30] combines hash table with B^+-Tree. However, HiKV suffers from high DRAM consumption and recovery cost.

One observation about the existing PM index structures is that they often perform many expensive *persist* primitives to maintain their structural properties or update metadata. For example, in FASTFAIR[12], an insert to a leaf requires, on average half cache lines to be persisted to keep the node sorted and durable. FPTree [23] needs to persist the fingerprints and bitmap in the leaf node per update. However, only the inserted key-value requires a round trip to PM for durability. We refer to such additional writes to PM as **structural maintenance overhead** (SMO).

Motivated by this observation, we set out to design a persistent index structure that reduces SMO. Also, we aim to reduce store fencing instructions as well since fencing drains CPU store buffer and stalls pipeline [9]. We notice that if we could perform index updates in batches, then persistence overhead could be amortized. For example, a batch of

*corresponding author

Proceedings of the VLDB Endowment, Vol. 13, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3372716.3372717

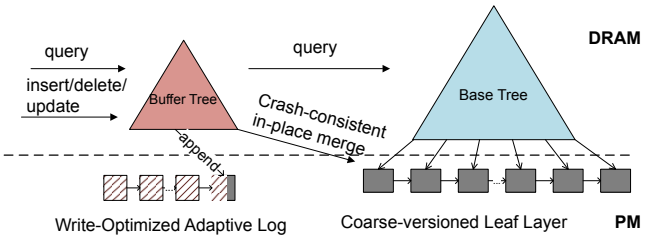


Figure 1: DPTree Architecture

key-value entries could be inserted into a leaf node in PM with only one update to the metadata. Based on this idea, we design a two-level persistent index partly inspired by the *dual-stage index architecture* [32]. The first level (called **buffer tree**) is in DRAM and the second level (called **base tree**) is in PM. Writes are first absorbed by the fast and small buffer tree backed by a write-optimized PM redo log for durability. When the buffer tree reaches a size threshold, we exploit the byte-addressability of PM to merge the buffer tree into the base tree **in-place**, and the changes are then flushed to PM with little fencing overhead. This way, the SMO is amortized over a batch of updates. Besides, the base tree is heavily read optimized to reduce the search latency introduced by the two-level design. Experiments show that DPTree reduces PM writes by a factor of 1.7x-3x compared to state-of-the-art counterparts and meanwhile maintains a comparable or better search latency. We summarize the contributions as follows:

- We present DPTree, which consists of the following designs: **1)** A write-optimized adaptive PM log for the buffer tree. **2)** A novel coarse-grained versioning technique for the base tree that provides crash-consistency and meanwhile enables concurrent read. **3)** An in-place crash-consistent merge algorithm based on the versioning technique that reduces writes. **4)** A hashing-based in-PM leaf node design that reduces PM reads while preserving efficient range scan.
- Apart from the single-threaded DPTree, we propose additional designs to make DPTree concurrent and scalable in terms of read, write, parallel merge, and parallel recovery.
- We show that DPTree achieves superior performance in terms of read and write compared to state-of-the-art structures in our experiments on realistic workloads.

The rest of the paper is organized as follows: Section 2 details the design of DPTree. Section 3 discusses the concurrent version of DPTree. Section 4 discusses the experimental results. Section 5 presents the related work. Finally Section 6 concludes the paper.

2. DIFFERENTIAL PERSISTENT TREE

As Figure 1 shows, DPTree consists of two tree structures, namely *buffer tree* and *base tree*. The *buffer tree* is DRAM-resident and is optimized for updates. We design an adaptive write-optimized logging scheme for the buffer tree for durability. With this log, writes of key-value payload whose log entry fits in a cache line to the buffer tree require only one `persist` to the PM. The *base tree* is read-only and employs *selective persistence* scheme used in [31, 23] where the DRAM-resident part is a radix tree while the PM-resident part is a linked list of leaf nodes. Inserts/updates are implemented as upserts to the buffer tree. If the key is not in the buffer tree, it is inserted into the tree. Otherwise, its value is overwritten. Deletions are implemented as upserts as well using *tombstone record*: one bit of value is reserved to indicate whether the corresponding key is deleted. Point query first searches the buffer tree. If the key is found, the query returns the latest value. Otherwise, the base tree is consulted. For range queries, both trees are searched, and results are merged.

DPTree maintains a merge threshold $R \in (0, 1)$ for the size ratio between the buffer tree and the base tree (e.g., 1:10). When the size ratio exceeds the threshold, a merge process starts. We discuss the merge process in detail in section 2.3. As an optimization, we maintain for the buffer tree a bloom filter similar to the *dual-stage index architecture* [32] to filter out searches for keys that are not in the buffer tree.

2.1 Buffer Tree

The buffer tree is an in-DRAM B$^+$-Tree whose durability is guaranteed by a PM write-ahead log. Since buffer tree is always rebuilt on recovery, the log only stores the logically upserted key-value entries which usually results in less storage footprint than traditional redo/undo logging for page-based storage.

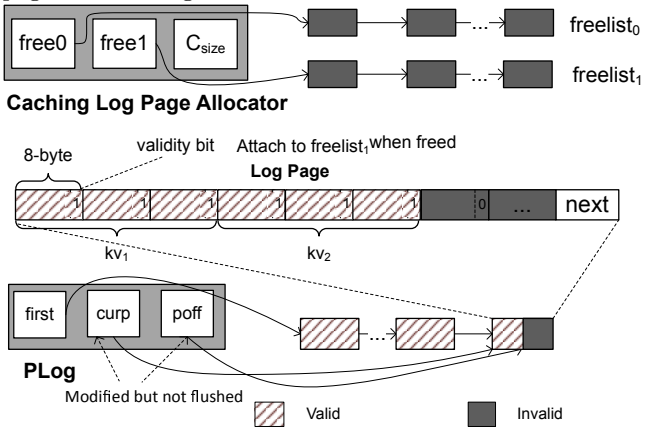


Figure 2: Buffer Tree Log Layout

Write-Optimized Adaptive Logging We assume that the key-value entries are fixed-length. Our goal is to update the B$^+$-Tree durably using a PM log. One naïve way to implement such a log is to maintain a persistent linked list of log records. This requires at least two `persist`s for every append operation: one for the log record and one for the link pointer. Instead, we use an idea similar to the Mnemosyne system [27], leveraging the 8-byte failure-atomic write in modern hardware. First, a log buffer is allocated, and one bit is reserved for every 64-bit word as the validity bit. We initially set the validity bit of every word to be 0. We serialize a log record (key-value entry) into multiple 64-bit words, each holding 63-bit information and a validity bit of 1. These words are then written into the log buffer and persisted. When an entry fits in a cache line after serialization, this log-append requires only one `persist`. When reading the log on recovery, we can detect torn writes by checking the validity bits of log records. One flaw of this method is that it requires initialization, doubling the number of PM

writes. Mnemosyne proposes to set the buffer fix-sized and reuse the buffer after it is full by flipping the meaning of the validity bit(e.g., 1 for invalid and 0 for valid). However, this would not work in our case as the buffer tree could grow in capacity. Hence the log buffer needs to grow as well. To solve this problem, we organize the log as a persistent list of log pages and employ a *caching* allocation scheme. As shown in Figure 2, the linked list includes a persistent header *PLog*, which includes a persistent pointer to the first page, a volatile pointer to the current page and a volatile offset within the current page. Our log buffer allocator maintains two freelists of pages, namely $freelist_0$ and $freelist_1$: one with records whose validity bits are set to 0 and the other set to 1. When a buffer tree is created, its log chooses one of the two freelists for allocation based on a boolean-valued version number that is flipped after each merge. When the chosen freelist runs out of pages upon allocation request, it is filled with pages from the system allocator. After a merge for the buffer tree completes, the full pages in the chosen freelist are transferred to the other freelist for the next buffer tree. Using this allocation scheme, we can reuse most of the existing pages and pay the persistence overhead only when the cached free pages are in shortage. To keep freelist size reasonable when index shrinks due to deletions, we dynamically maintain the cache capacity C_{size} to be about the size of the buffer tree before triggering a merge. Since the buffer tree grows or shrinks at the rate of R at most, the cache size adapts to this rate as well. To amortize the persistence overhead introduced by the maintenance of the freelists, we can increase the page size.

2.2 Base Tree

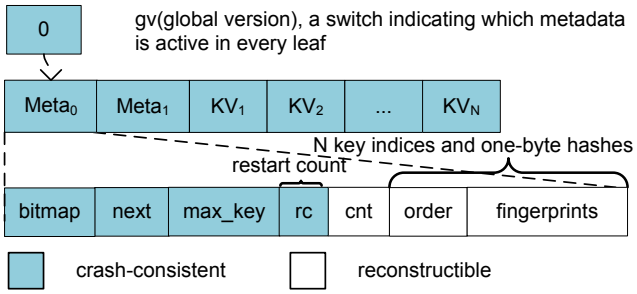


Figure 3: Base Tree Leaf Layout

In DPTree, the base tree is read-only and updated only by merge. Therefore we optimize it in terms of read and merge. To take advantage of DRAM-PM architecture, we employ *selective persistence* scheme in the base tree where the internal nodes are in DRAM while the leaves are in PM. Two consecutive leaves are relatively ordered while the leaf itself may not, and the leaf list is *coarse-grained* versioned for persistence.

Coarse-Grained Versioning Leaves. The CDDS B-Tree [26] proposes to version key-value entry at every write operation to achieve crash consistency. However, such fine-grained versioning incurs large overheads. Inspired by the versioning technique, we make the following observation: ***the before-merge and after-merge image of the base tree provide a natural versioning boundary***. Based on this observation, we propose the coarse-grained versioning technique. We maintain in every leaf node two sets of metadata and a key-value slot pool, as shown in Figure 3. Each metadata consists of a bitmap, a next pointer, the max key in the node, the number of key values in the node, an order array, and an array of fingerprints. The bitmap tracks the allocation status of the key-value slots. The order array stores the ordered indices to the key-value slots similar to wB$^+$-Tree [9] to accelerate range scan. The key-value pool is organized by hashing, and collisions are resolved using *linear probing*. The *fingerprints* array stores one-byte hashes of the keys stored at the corresponding positions of the key-value pool. It serves to filter out unnecessary PM reads. To support deletion in linear probing hash table, we use two bits in the bitmap for each slot: *00* for *empty* slot, *01* for *occupied* slot, and *10* for *deleted* slot. Hash probing continues until either a matching key or an empty slot is found. A persistent global version number *gv* is maintained to indicate which metadata is active in every leaf node. *gv* also denotes the freelist for the current buffer tree discussed in section 2.1. Merge handles updates to a key-value entry out-of-place but mostly within a leaf. That is, we find an empty/deleted slot within the leaf and update the inactive metadata. When the merge completes, *gv* is flipped. Therefore *gv* combined with dual metadata distinguishes the before-image from the after-image of the leaf list with low overhead. Queries first get a snapshot of *gv* and traverse from the DRAM radix tree down to a specific leaf node. In the leaf node, the keys are examined with the help of the metadata of the snapshot version.

Partial Persistence. One interesting aspect to note is that the leaf node is kept partially persistent. That is, the *count*, *order*, and *fingerprints* arrays are modified on merge but not persisted. We choose this design based on the observations that system failures are rare, and these auxiliary data could be rebuilt from the rest of the crash-consistent data. We do not rebuild these metadata immediately at restart as this slows down recovery. Instead, the rebuilding is completed *cooperatively* by query or merge after the restart. Specifically, we maintain a persistent global counter *restartCount* to indicate the number of restarts. Inside every leaf node metadata, we keep a local restart count. At restart, *restartCount* is incremented persistently. When a query or merge locates a leaf, the local count is checked against the global one. If they do not match, the metadata is rebuilt, and the local restart counter is updated persistently.

2.3 Crash Consistent Merging

Since the base tree is in PM, the merge needs to be crash-consistent. A straightforward way to implement merge is similar to LSM [22]: merge buffer tree and base tree into a new base tree. However, this method has a write amplification of $\frac{1}{R}$ in the worst case. To maintain fast recovery, typically, the merge threshold R is chosen to be a small number, e.g., 0.1, resulting in high write amplification. The source of write amplification is the out-of-place merge style optimized for HDD/SSD but not for PM. Therefore, we propose to reduce writes by exploiting the byte-addressability of PM and merging the buffer tree into the base tree in-place.

To maintain crash-consistency during merge, we exploit the coarse-grained versioning technique and follow a simple principle – never modify the data of the current version indicated by the global version number *gv*. As shown in Algorithm 1, our merge algorithm consists of 7 phases. Given

Algorithm 1: BaseTreeMerge(tree, sit, eit)

input : $[sit, eit)$: a pair of iterator ranging the ordered entries from buffer tree

```
1  cv ← get_version(tree.gv), nv ← ¬get_version(tree.gv)
2  set_bit_and_persist(tree.gv, in_merge)
3  /* phase-1: upsert merge */
4  prevLeaf ← tree.head[cv]
5  curLeaf ← prevLeaf.meta[cv].next, it ← sit
6  while curLeaf != null do
7  |  E_curLeaf ← non-tombsone entries in [it, eit) whose key ≤ curLeaf.meta[cv].max_key, moving it accordingly
8  |  prevLeaf ← LeafUpsertMerge(prevLeaf, curLeaf, cv, nv, E_curLeaf)
9  |  curLeaf ← curLeaf.meta[cv].next
10 end
11 - create leaves after prevLeaf.meta[nv].next if it! = eit
12 /* phase-2: delete merge */
13 curLeaf ← tree.head[nv].meta[nv].next, it ← sit
14 while curLeaf != null ∧ it != eit do
15 |  E_curLeaf ← non-upsert entries in [it, eit) whose key ≤ curLeaf.meta[nv].max_key, moving it accordingly
16 |  LeafDeleteMerge(curLeaf, nv, E_curLeaf)
17 |  curLeaf ← curLeaf.meta[nv].next
18 end
19 /* phase-3: consolidation */
20 - For leaves of version nv with occupancy < T ∈ (0, 0.5], merge them with neighboring leaves
21 /* phase-4: flush to PM */
22 For each leaf of version nv, flush cache lines containing newly inserted entries and the crash consistent metadata without using any fences.
23 /* phase-5: rebuild volatile internal nodes */
24 /* phase-6: flip the global version number */
25 set_and_persist(tree.gv, nv)
26 /* phase-7: garbage collection */
27 - Delete leaves that can be reached through tree.head[cv] but not through tree.head[nv]
```

Algorithm 2: LeafUpsertMerge(prevLeaf, curLeaf, cv, nv, E_{curLeaf})

input : E_{curLeaf}: ordered key-value entries whose key ≤ curLeaf.meta[cv].max_key

output: last leaf after merge

```
1  curLeaf.meta[nv] ← curLeaf.meta[cv]
2  if curLeaf.meta[nv].rc != restartCount then
3  |  - reconstruct metadata of version nv
4  |  curLeaf.meta[nv].rc ← restartCount
5  freeslots ← LeafCapacity - curLeaf.meta[nv].count
6  if kvs.length ≤ freeslots then
7  |  - Upsert entries by hashing and merging using bitmap and order array of version nv
8  |  return curLeaf
9  else
10 |  NL ← create leaves filled with entries from merging E_curLeaf with curLeaf at filling rate FR
11 |  prevLeaf.meta[nv].next ← NL.first
12 |  NL.last.meta[nv].next ← curLeaf.meta[cv].next
13 |  return NL.last
14 end
```

Algorithm 3: LeafDeleteMerge(leaf, nv, kvs)

input: kvs: ordered key-value entries whose key ≤ leaf.meta[nv].max_key

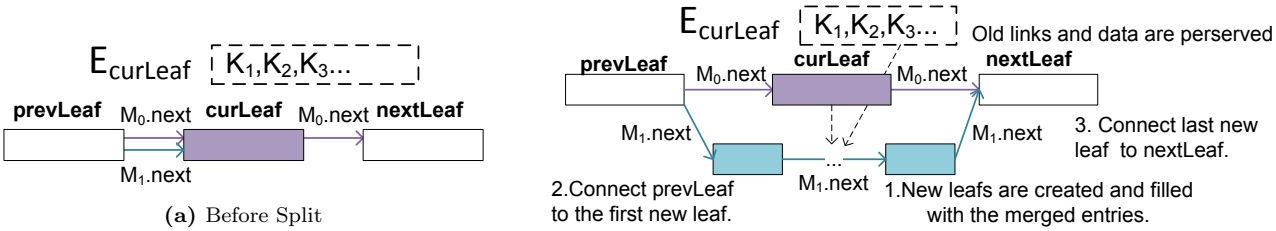
```
1  i ← 0, j ← 0, leafKeyCount ← leaf.meta[nv].count
2  while i <leafKeyCount ∧ j <kvs.length do
3  |  leafKeyIdx ← leaf.meta[nv].order[i]
4  |  leafKey ← leaf.kvs[leafKeyIdx].key
5  |  if leafKey <kvs[j].key then
6  |  |  i++
7  |  else if leafKey = kvs[j].key then
8  |  |  leaf.meta[nv].bitmap[leafKeyIdx] ← deleted
9  |  else
10 |  |  j++
11 |  end
12 end
13 - reconstruct meta[nv].order and meta[nv].count
```

a pair of iterator ranging the ordered key-value entries from the buffer tree, our merge algorithm starts by getting the current version *cv* from the global version number *gv* and computes the next version *nv*(line 1). We then persistently set the *in_merge* bit of *gv* to indicate the presence of a merge (line 2) which is used in recovery. Then the algorithm executes the following 7 phases.

Upsert Merge Phase. We enumerate the leaves of version *cv*(line 4-11) to perform the merge. For every leaf denoted as *curLeaf*, we get the entries to be merged into *curLeaf* by extracting from the input sequence the entries E_{curLeaf} whose keys are no greater than the max key of *curLeaf* (line 7). Most of the heavy-lifting of the merge between *curLeaf* and E_{curLeaf} is handled by *LeafUpsertMerge*(line 8). As shown in Algorithm 2, it first makes a copy of *meta[cv]* into *meta[nv]*. Then it rebuilds metadata of version *nv* if the local restart counter does not match the global one(line 2-3) and updates the local restart counter. If there are enough free slots for the incoming entries, they are merged using the order array and bitmap in *meta[nv]*(line 6-8). An illustration of such a case is shown in Figure 5. Otherwise, node split is needed. Figure 4 shows a before and after image of a node split. We first create as many new leaves as needed to fill in the merged entries between *curLeaf* and E_{curLeaf}. Then new leaves are linked using metadata of version *nv* in *prevLeaf*. Since we only modify the metadata of the inactive version *nv* and the unused key-value slots in each leaf, even if the system crashes during the split, we are still able to recover the data structure via the links of version *cv*. In addition, we do not fill the new leaves fully, as this might result in excessive splits in future merges. This filling rate is denoted as a parameter $FR \in (0.5, 1]$.

Delete Merge Phase. Deletion is similar to upsert. The difference is that deletion only needs to modify the bitmap. As shown in line 13-18 of Algorithm 1, E_{curLeaf} is extracted from the merge sequence for each leaf node and *LeafDeleteMerge* is called. In *LeafDeleteMerge*(Algorithm 3), deleted entries are first found similar to merge sort. Then the bits of deleted entries in the bitmap are marked(line 8). Lastly, the order array and count of version *nv* are reconstructed using the updated bitmap and entry pool.

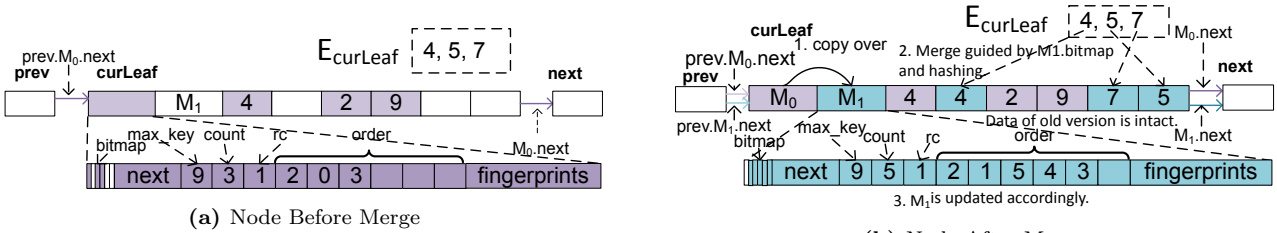
Consolidation Phase. To maintain a reasonable node



(a) Before Split

(b) After Split

Figure 4: Before and After Image of Leaf Upsert Merge(Split)



(a) Node Before Merge

(b) Node After Merge

Figure 5: Before and After Image of Leaf Upsert Merge(No Split)

occupancy, leaves of version *nv* having occupancy below a consolidation threshold $CT \in (0, 0.5]$ are merged with adjacent nodes. They are merged into the next leaf if there are enough free slots. If not, entries are borrowed from the next leaf until these two leaves have about the same occupancy.

Flushing Phase. As shown in line 22-25 of Algorithm 1, we use flush instruction on every leaf of version *nv* to flush cache lines that contain newly inserted entries. We skip cache lines that contain only newly deleted entries because deletion requires only the bitmap to be flushed. Then the crash-consistent metadata are flushed to PM. Notice we do not use fences to order these flushes because the one *mfence* in Global Version Flip phase will guarantee these writes are persisted to PM, reducing the need for fences significantly.

Rebuilding Internal Nodes. Then the internal nodes are rebuilt from the leaves of version *nv* using the max_key and persistent address of each leaf. As our leaf nodes are updated infrequently, maintaining the max key is very cheap. Moreover, keeping the max key also speeds up recovery as the amount of data read from PM is drastically reduced.

Global Version Flip. Once all the above phases finished, the global version number *gv* is flipped and persisted to PM. This officially concludes the merge and in effect, frees up the space occupied by the old entries for the next merge.

Garbage Collection. During the above phases, garbage leaves might be generated and, therefore, require reclamation. For example, leaf split renders the old leaf obsolete. The identification of garbage nodes is simple. We denote L_{cv} as the set of leaves obtained through the links of version *cv*. Similarly L_{nv} is obtained through the links of version *nv*. Then the garbage leaves are simply $L_{\text{cv}} - L_{\text{nv}}$. Note that garbage collection is placed after the version flip. This is because only when the flipped global version reached PM, can we safely reclaim space. To handle crashes during reclamation, we traverse the leaves of version *cv* backwards and delete every node *garbageLeaf* that is in $L_{\text{cv}} - L_{\text{nv}}$ using the address stored in the predecessor of *garbageLeaf*. In this way, we can re-execute the reclamation during recovery freely.

2.4 PM Management & Recovery

PM Allocation. The non-volatility of PM introduces challenges for memory allocation. For example, the system could crash in between a region allocated from PM allocator, and the address of that region is persistently stored in the user's data structure, resulting in PM leaks. To avoid this, we assume that the system is equipped with PM allocator with following APIs [24]: `reserve`, `activate`, and `free`. The allocator first reserves a region of PM per user request via the `reserve` API. The user then provides a PM location to persistently receive the address of the reserved memory region through the `activate` API. Only after the `activate` returns can we consider the allocation complete. On deallocation, the user provides the same PM location to the `free` API. The allocator then failure-atomically frees up the memory region and sets the PM location to contain a pre-defined value(e.g., null). This ensures that user programs can avoid double freeing the memory region by checking the persistent pointer against the pre-defined value.

Algorithm 4: Recover(tree)

```
1 in_merge ← extract_in_merge_bit(tree.gv)
2 if in_merge = 1 then
3 |   - reclaim new leaves created during the merge.
4 |   clear_bit_and_persist(tree.gv, in_merge)
5 - finish garbage collection
6 - replay PLog sequentially to reconstruct the buffer
    tree.
7 inc_and_persist(restartCount)
```

Recovery. On system restart, the index needs to recover to a consistent state. The recovery procedure shown in Algorithm 4 first determines if the crash happened during merge by examining the *in_merge* bit of the global version number *tree.gv*. If the bit is set, leaf nodes created during the interrupted merge require garbage collection. To be able to reclaim them, we persistently store the address of every new leaf in a persistent linked list *tree.pendingLeaves* before linking them in the leaf list during merge. Upon successful merge, the list *tree.pendingLeaves* itself is destroyed persistently using the *free* API of the system allocator. When an interrupted merge is detected, we reclaim those leaves by traversing *tree.pendingLeaves* and deleting pending leaves

using addresses stored in the list node through *free* API of the system allocator. Then the algorithm continues the GC process if needed. This includes re-run of the garbage reclamation and persistently destroying the *tree.pendingLeaves* itself. Then the buffer tree is rebuilt from PLog. At the last step, *restartCount* is incremented.

2.5 Managing Complexity

It might seem complicated to equip each index structure with a standalone log, though such approach is used in several works [9, 23]. However, the standalone log is essential for providing durability with low overhead. One might consider reducing the complexity by keeping a single log for multiple index instances. However, multiple instances might interfere with each other and trigger merges unnecessarily, and therefore increase the complexity. Instead, one could hide the complexity via encapsulation since DPTree provides the same interfaces as other indices, which makes integration with real systems easier.

3. CONCURRENT DPTREE

Multi-core architectures are the pervasive now. To unleash the full potential of multi-core processors, efficient design of concurrent operations on persistent index structures is crucial. Therefore, we address the design of concurrent DPTree.

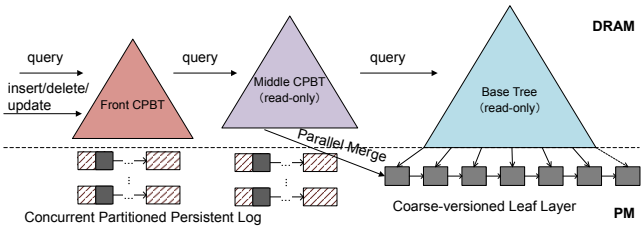


Figure 6: Concurrnet DPTree Architecture

The concurrency scheme for DPTree can be described in three parts: the concurrent persistent buffer tree(CPBT), parallel merge, and the synchronization between tree components. Figure 6 shows the architecture of concurrent DPTree. Concurrent writes are handled by the *Front CPBT*. When the *Front CPBT* is full, it becomes read-only and named as *Middle CPBT*. A new CPBT is created to be the *Front CPBT* to serve new writes. The base tree then merges with the *Middle CPBT* in parallel in the background. Reads consult in sequence of *Front CPBT*, *Middle CPBT*(if any), and finally *base tree*. Reads finish when a matching key is found in any of the three components.

3.1 Concurrent Persistent Buffer Tree

The concurrent persistent buffer tree consists of an in-DRAM concurrent B$^+$-Tree and a PM log. Since there are many studies on concurrent DRAM indices, such as Masstree [19], BwTree [17] and ART [16], we employ OLC (Optimistic Lock Coupling) [16] in our B$^+$-Tree as the synchronization mechanism for its simplicity and efficiency. The main challenge, however, is to make the log persistent, concurrent, and scalable for recovery.

Concurrent Logging We solve this by hash partitioning the log on key into N smaller partitions. Each partition is implemented as a concurrent persistent linked list of log pages. And each partition is equipped with a caching page

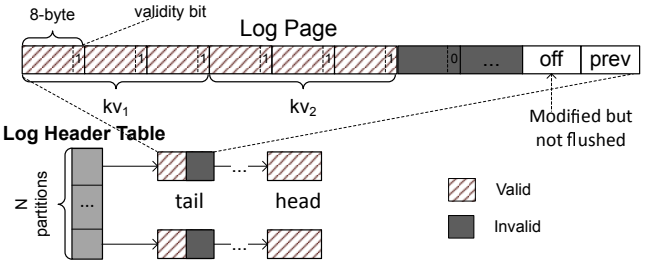


Figure 7: Concurrent Buffer Tree Log Layout

Algorithm 5: ConcurrentLogAppend(tree, r)

```
1  p ← hash(r.key) % N
2  restart: tailp ← tree.log_headers[p]
3  goto new_page if tail is null
4  off ← FetchAndAdd(tailp.off, sizeof(r))
5  if off + sizeof(r) ≤ StorageAreaCap then
6  |  - write and persist r into tailp.storage at offset
   |    off
7  else
8  |  new_page: lock partition p
9  |  newp ← callocator.reserve(tree.gv)
10 |  newp.off ← 0, set_and_persist(&newp.prev,
   |    tailp)
11 |  callocator.activate(&tree.log_headers[p], newp)
12 |  - unlock partition p
13 |  - atomically increment the size of buffer tree by
   |    the number of log records per page
14 |  goto restart
```

allocator as well. As can be seen from Figure 7, the log has a header table of size N with each slot storing the address of the tail page of each list. The next log record will be written into the tail page. Each log page is fixed-size consisting of metadata and a storage area. The *off* in metadata is the writing position of the next log record within the storage area. The *prev* pointer points to the previous log page. A page with null value in *prev* field indicates the head of the linked list. Therefore, the recovery procedure starts from this head page and walks back to the tail page. To reduce PM writes, *off* is only modified but not persisted. This is tolerable because, upon recovery, valid records could be found by examining the validity bits in each log record.

The *append* algorithm for the log is shown in Algorithm 5. To append records concurrently, writers find the tail page of a partition through the header table and use an atomic FAA(fetch-and-add) instruction on the *off* field to claim the writing position and space for its record. If the claimed offset is within the capacity of the log page, the record is written and persisted(line 6). Otherwise, a new log page is allocated to be the new tail page for a partition. A partition lock protects this allocation. The caching allocator is modified to support the two-step (i.e., reservation + activation) allocation scheme. Though we use locks here, it turns out not to be a bottleneck in our experiments. This is because the hot code path is the log record writing, which is already lock-free. Finally, we update the size of the buffer tree at the end of a page allocation (line 13). This, in effect, makes the buffer tree size approximate and prevents the shared size counter from becoming a bottleneck.

Concurrent Modification. We now describe the algorithm of CPBT upsert (*insert*, *delete*, and *update* are all implemented as upsert). To upsert key X, a log record is written first before the B$^+$-Tree is modified. However, the order of records of X in the log must match the order of modifications of X applied to the B$^+$-Tree. This implies that the log appending and tree modification for an upsert operation should be an atomic operation. Consider the case where one thread is inserting key X, and another is deleting X. If these two steps were not an atomic operation, it would be possible that in the log, the deletion of X is ordered before the insertion of X while in the B$^+$-Tree the insertion is ordered before the deletion. On crash recovery, the reconstructed B$^+$-Tree will be different from the one before the crash. To prevent this, we utilize the write lock in the leaf node of the Optimistic Lock Coupling B$^+$-Tree. On every upsert, we first acquire the lock in the B$^+$-Tree leaf, then concurrent logging is performed, followed by modification of the leaf, and finally, bloom filter is updated. This correctly serializes conflicting upsertions of the same key .

3.2 Synchronizing Tree Components

Algorithm 6: ConcurentSearch(key)

```
1  - seq-cstly set W_FR.active
2  W_FR.ptr ← GP_F
3  - search in W_FR.ptr
4  W_FR ← 0
5  - return value if found
6  - seq-cstly set W_MR.active
7  W_MR.ptr ← GP_M
8  - search in W_MR.ptr if not null
9  W_MR ← 0
10 - return value if found
11 - seq-cstly set W_BR.active
12 W_BR.ptr ← GP_B
13 - search in W_BR.ptr
14 W_BR ← 0
15 - return value if found
```

Concurrent DPTree has multiple components that need to be synchronized. For eaxmple, after tree merge, GC should be performed only when there are no readers that have references to the *Middle CPBT*. This essentially requires tracking the number of active reader/writer. One naïve way is to use globally shared counters [11]. However, previous studies [8, 19] have shown that frequent writes to shared cache lines kill scalability due to high cache invalidation traffic. Instead, we choose to track the reader/writer in a decentralized way similar to RCU [20]. We maintain four thread-local synchronization words: $\boldsymbol{W_{FW}}$, $\boldsymbol{W_{FR}}$, $\boldsymbol{W_{MR}}$, and $\boldsymbol{W_{BR}}$. $\boldsymbol{W_{FR}}$/$\boldsymbol{W_{FW}}$ stores the address of the *Front CPBT* current reader/writer thread is accessing. $\boldsymbol{W_{MR}}$ stores the address of the *Middle CPBT* current reader thread is accessing. $\boldsymbol{W_{BR}}$ stores the root of the radix tree in base tree current reader thread is accessing.

One bit of each word is reserved as *active* bit indicating the presence of a reader/writer. The rest of the bits store a pointer value, acting as a reference. All threads register the addresses of the words to a synchronization manager(SM) at startup or first access. We also maintain three global pointers: $\boldsymbol{GP_F}$, $\boldsymbol{GP_M}$, and $\boldsymbol{GP_B}$ pointing to current *Front CPBT*, *Middle CPBT* and the root of the radix tree in *base tree* respectively. We assume accesses to shared word-sized values are atomic with acquire/release semantics(e.g., x86).

Algorithm 7: ConcurrentUpsert(key, value)

```
1  - seq-cstly set W_FW.active
2  W_FW.ptr ← GP_F
3  - upsert (key, value) into W_FW.ptr
4  FrontCPBTSize ← W_FW.ptr.size
5  W_FR ← 0
6  if FrontCPBTSize ≥ MergeThresh then
7      - spin while MergeState = premerge
8      if CAS(MergeState, nomerge, premerge) then
9          F' ← create a new CPBT
10         F ← GP_F
11         - seq-cstly set GP_M with F
12         - seq-cstly set GP_F with F'
13         SpinUntilNoRef(TYPE_W_FW, F)
14         MergeThresh ← GP_B.size * R
15         - seq-cstly set MergeState with merging
             state
16         - spanw a new thread to run following code:
17          - ParallelTreeMerge(GP_M, GP_B)
18          - MP ← GP_M
19          - GP_M ← 0
20          - SpinUntilNoRef(TYPE_W_FR, MP)
21          - SpinUntilNoRef(TYPE_W_MR, MP)
22          - delete MP
23          - seq-cstly set MergeState with nomerge
              state
```

Algorithm 8: SpinUntilNoRef($WType, ptr$)

```
1 while True do
2     Wait ← false
3     for each thread t registered in SM do
4         W ← load sync word of t of type WType
5         if W.active ∧ (W.ptr = 0 ∨ W.ptr = ptr)
            then
6             Wait ← true
7     end
8     - break if Wait = false
9 end
```

Tracking Readers. The steps for concurrent search is shown in Algorithm 6. It starts by sequential consistently (abbreviated as seq-cstly) (e.g., using `mfence` in x86) setting the active bit of $\boldsymbol{W_{FR}}$, declaring the entrance of a reader. Then, the $\boldsymbol{GP_F}$ is loaded into other bits of $\boldsymbol{W_{FR}}$, acting as a reference to the data structure. Then the search is performed on *Front CPBT*. Finally the $\boldsymbol{W_{FR}}$ is cleared, declaring the exit of the reader. The access to *Middle CPBT* and base tree is guarded similarly.

Synchronization for Metadata Rebuild. The concurrent search in base tree is similar to the single threaded search because we never modify the data of previous version during merge. However, after restart, the metadata in a leaf might require a rebuild which needs synchronization. This is achieved by using one bit in the local restart counter of each leaf as lock bit. During search or merge, the local restart counter is checked against the global one. If they do not match, the lock is obtained and the metadata is rebuilt. To prevent persistent deadlocks, we clear lock bits on recovery.

Tracking Writers & Merge Initiation. The steps for

concurrent upsert is shown in Algorithm 7. It employs similar guarding mechanism during the write to *Front CPBT*. After the write, if the size of *Front CPBT* is full, a merge task is scheduled by only one writer. This is achieved by atomically manipulating the $MergeState$ integer which has three states that transitions in a cyclic order: *nomerge*, *premerge*, and *merging*. To ensure there is at most one on-going merge, threads spin until the $MergeState$ is not *premerge* before CAS'ing it to *premerge*. The winner gets to initiate a merge task. It first sets $\boldsymbol{GP_M}$ to the full *Front CPBT* F, allowing new readers to perform reads on the temporary buffer tree. Then $\boldsymbol{GP_F}$ is set to a new CPBT F'. Notice there might a short period of time where $\boldsymbol{GP_F} = \boldsymbol{GP_M}$ (after line 11 before line 12). In such a case, duplicate reads are possible. However, it is still correct.

After the $\boldsymbol{GP_F}$(line 12) is updated, there still might be writers left accessing the CPBT indicated by $\boldsymbol{GP_M}$. We must wait for these writers to finish before the $\boldsymbol{GP_M}$ is used for merge(line 13). This is done by scanning the $\boldsymbol{W_{FW}}$ in each threads and spins as long as there are possible writers left: there exists a $\boldsymbol{W_{FW}}$ where the *active* bit is on and the pointer value stored in the word equals to null or $\boldsymbol{GP_M}$. The test of *active* bit excludes non-writer threads. The test of null handles the race condition where writer threads have loaded the value of $\boldsymbol{GP_F}$ but have not stored the value into $\boldsymbol{W_{FW}}$.ptr. Notice the sequential-consistent stores to the thread-local words are crucial as they prevent the loads of global pointers being reordered before the stores to the active bits, which might result in missing references during checks. The last condition catches the writers currently accessing $\boldsymbol{GP_M}$. Since this synchronization mechanism is used in multiple places, we abstract it into an general procedure $SpinUntilNoRef$ shown in Algorithm 8 which takes as input a enum (possibly valued $\boldsymbol{W_{FW}}$, $\boldsymbol{W_{FR}}$, $\boldsymbol{W_{MR}}$, or $\boldsymbol{W_{BR}}$) and a pointer value. Then we computes the next merge threshold (line 14), and $MergeState$ is changed to *merging* state(line 15). Finally a background thread is spawned to execute the merge task.

In the merge worker, $ParallelTreeMerge$ is called to perform the real merge (see next section). When the merge completes, the *Middle CPBT* should be destroyed to accelerate reads. Similarly, there are possible readers left accessing $\boldsymbol{GP_M}$. As shown in line(17-23) of Algorithm 7, after the merge, we first set $\boldsymbol{GP_M}$ to null atomically, ensuring new readers go directly to the base tree. Then we use the procedure $SpinUntilNoRef$ on $\boldsymbol{W_{FR}}$ and $\boldsymbol{W_{MR}}$ of each thread to wait for the readers to exit before the CPBT, indicated by $\boldsymbol{GP_M}$, is destroyed. Notice we scan $\boldsymbol{W_{FR}}$ as well because, though unlikely, a reader might already be reading the *Front CPBT* before the *Front CPBT* becomes the *Middle CPBT* and finish its read after the merge has completed. Lastly, the old *Middle CPBT* is destroyed and $MergeState$ is updated.

3.3 Parallel Tree Merging

To reduce the blocking time introduced by base tree merge, parallel merging is essential. Luckily, the merge in our case is easy to parallelize. As shown in Algorithm 9, we divide the leaves of base tree into partitions, denoted as $P_1 ... P_{\mathrm{M}}$. To be more specific, each partition P_{i} contains a constant $PSize$ number of consecutive leaves of base tree denoted by two endpoint leaves: $P_{\mathrm{i}}.BLS$ and $P_{\mathrm{i}}.BLE$. We also denote $min_{\mathrm{key}}(L)$ and $max_{\mathrm{key}}(L)$ as the smallest and greatest key in leaf L. For each partition

Algorithm 9: ParallelTreeMerge(MT, BT)

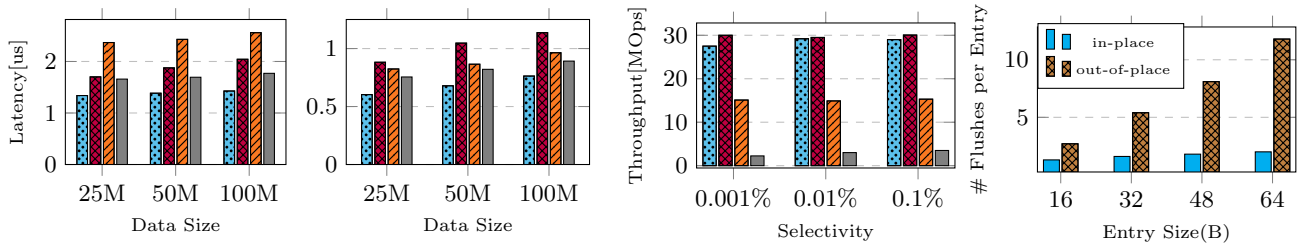
```
1  P_1..P_M ← partition BT and MT into N
     independent merge partitions
2  /* pull-based task allocation */
3  for each merge worker thread in the system do
4  |  p ← get next to-be-merged partition in P_1..P_N
5  |  - perform UpsertMerge, DeleteMerge and
   |    Consolidation Phases on p
6  end
7  - wait for all partitions to be processed
8  - reconnect consecutive partitions in P
9  for each merge worker thread in the system do
10 |  p ← get next to-be-flushed partition in P_1..P_N
11 |  - perform Flush Phase on p
12 end
13 - rebuild the radix tree and update GP_B atomically
14 - wait for all partitions to finish flushing and any
     readers accessing old radix tree to exit using
     SpinUntilNoRef
15 - perform global version flip
16 - perform garbage collection with one thread
```

$P_{\mathrm{i}} (1 \leq i \leq M)$, we find in *Middle CPBT* the first key $P_{\mathrm{i}}.MKS \geq min_{\mathrm{key}}(P_{\mathrm{i}}.BLS)$. Similarly, we find in *Middle CPBT* the first key $P_{\mathrm{i}}.MKE > max_{\mathrm{key}}(P_{\mathrm{i}}.BLE)$. The key range denoted by $[P_{\mathrm{i}}.MKS, P_{\mathrm{i}}.MKE)$ in *Middle CPBT* is to be merged into leaves denoted by $[P_{\mathrm{i}}.BLS, P_{\mathrm{i}}.BLE]$ in base tree. Since these partitions are non-overlapping, they could be merged in parallel without any synchronization.

These merge partitions are then scheduled to a pool of worker threads via a queue to balance the load on each worker. Moreover, the $PSize$ is set to be large enough to amortize the scheduling overhead but also small enough to prevent each merge task to become a bottleneck of the entire merge. The specifics of the merge is similar to Algorithm 1 with small modifications. The three phases UpsertMerge, DeleteMerge, and Consolidation are performed in parallel for all merge partitions (line 3-6). After these phases are done and synchronized, there might be leaves newly created or deleted in some partitions. Therefore, we connect the consecutive partitions through the next pointers(line 8). Then the Flushing Phase is also executed in parallel (line 9-12) along with a rebuilding of the internal nodes. Similarly, we perform the version flip and GC at the last step.

3.4 Optimizations for Word-Sized Value

When the value of the key-value entry is word-sized, and there is currently no merge (i.e., $MergeState = nomerge$), we exploit the 8B failure-atomic write to implement update/delete for keys not in the base tree without writing PM log. Assuming the key being updated/deleted is X, we first lock the leaf of the buffer tree where X will reside. Then we perform a search on the base tree for X. If X is found, its value is updated failure-atomically. For deletion, the tombstone bit is set, and X will be removed on the next merge. Otherwise, the update/delete operation reports that X does not exist. The update/delete operation follows the same guarding mechanism used previously. Specifically, it first obtains the reference to the *Front CPBT* in $\boldsymbol{W_{FW}}$ before checking $MergeState$. In this way, if $MergeState = nomerge$, we are sure that the merge between the *Front CPBT* to the base tree will not be initiated (because we held a reference) before the update/delete is



(a) Insert **(b)** Lookup **(c)** Scan **(d)** Merge Policy

Figure 8: Single-Threaded Micro-benchmark

complete, avoiding lost updates.

4. EVALUATION

In this section, we evaluate the DPTree against other state-of-the-art persistent index structures including FASTFAIR [12], FPTree [23], and WORT [15] on Intel Optane DC Persistent Memory Modules.

4.1 Implementation & Setup

We implemented the DPTree and its concurrent version in C++11. We utilize an open-source OLC B+Tree [1] in the buffer tree. For FASTFAIR and WORT, we use their public implementations [2]. Since FPTree is not open-sourced, we implemented it as faithfully as possible in C++11. For PM allocation, we use nvm_malloc [3] that supports the *reserve-activate* scheme. All source codes are compiled with g++6.4 with -O3. We ran the experiments on a Linux system (kernel 4.9) with two Intel(R) Xeon(R) Platinum 8263C CPU @ 2.50GHz (26 physical cores) CPUs with 35MB Last Level Cache, 376GB DRAM, and 512GB PM. The Optane DIMMs are configured in *App Direct* mode and exposed through ext4-DAX file system. To avoid NUMA effects, we bind all threads to one socket.

4.2 Micro-benchmark

In this experiment, we evaluate the performance of different index structures using a micro-benchmark. The keys are 64-bit random integers. For DPTree, we set the capacity of leaf to 256 entries, the merge consolidation threshold CT to 0.34, the merge threshold R to 0.1, the filling rate FR of base tree leaf to 0.7, log page size to 64KB. For FASTFAIR and FPTree, we use the default settings from their papers.

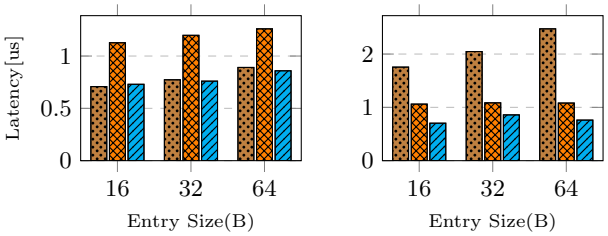
Insertion. We first insert key-value pairs of 16 bytes varying the data size from 25M to 100M and measure the average latency. The result is shown in Figure 8a. In general, DPTree has 1.27-1.43x lower latency than FASTFAIR and 1.76-1.78x than FPTree as the data size increases. We show the PM write statistics in Table 4.2. It can be seen that DPTree spends its flushes mostly in logging and node update. It also has the lowest number of total flushes/fences, which is translated into improved latency and endurance. Interestingly, WORT needs the most flushes and fences and incurs 1.25x higher latency than DPTree. This is because WORT requires key-value pair to be persistently allocated out-of-place, thus causing around 4 additional flushes. Whereas DPTree/FPTree/FASTFAIR show very low allocation overhead in Table 4.2 due to the page-based allocation.

Impact of Flush and Fence. To measure the performance impact of flush and fence, we first performed 50M insertions without flushes and fences. Then we gradually add back these instructions. We notice a 1.59/1.34/1.83/1.45x latency increase for DPTree/FASTFAIR/FPTree/WORT, respectively, when using only flush instructions. When both flush and fence instructions are added, we see an additional 1.10/1.12/1.10/1.13x increase. These results indicate that the impact of flushes is more significant than fences, and they both should be minimized.

Table 1: PM Flush Stats per op of 50M Insert Workload

Index	Sum	Log	Alloc	Node	Swap	Fence
DPTree	2.26	1.1	0.06	1.1	0	1.06
WORT	6.8	0	3.9	2.9	0	6.5
FASTFAIR	4.35	0	0.15	4.2	0	4
FPTree	3.64	0.05	0.07	3.51	0	3.6
2-Tier-BM	8.5	4	0.004	0	4.5	2

Point Lookup. We then perform random key searches varying data size from 25M to 100M. As shown in Figure 8b, DPTree is up to 1.82x/1.85x/2.19x faster than FASTFAIR, FPTree, and WORT. This is attributed to the efficient lookup inside base tree leaf and the bloom filter, eliminating most of the searches(95%) for keys not in the buffer tree. For PM-only indices, such as FASTFAIR and WORT, the cache misses to PM take up more overhead as the data size grows. Thanks to the buffer tree and inner nodes of the base tree being in DRAM, DPTree is able to complete the search mostly with 2 accesses to PM.



(a) Successful Lookup **(b) Unsuccessful Lookup**

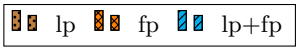


Figure 10: Comparision of Leaf Probing Schemes: **lp(Linear Probing), fp(Fingerprints), lp+fp(Linear Probing with Fingerprints)**

Range Scan. Next, we evaluate the performance of range scan of different indices. We measure the throughput of range scan on 50M data size while varying the scan selectivity from 0.001% to 0.1%. As shown in Figure 8c, DPTree

[1]https://github.com/wangziqi2016/index-microbench
[2]https://github.com/DICL/FAST_FAIR, https://github.com/SeKwonLee/WORT
[3]https://github.com/IMCG/nvm-malloc

is up to 3.0/10.98x faster than FPTree/WORT. This is because FPTree requires sorting, whereas DPTree avoids this cost using order array during scan. WORT performs the worst because it stores payloads randomly in PM, causing low cache utilization. In contrast, DPTree stores payloads inside a page for better locality. Compared to FASTFAIR, which is inherently good at scan, DPTree is only up to 1.1x slower due to the indirection through order array.

Space Consumption We then measure the memory utilization for all indices. After 50M insertions, the results are 1.7/1.26/1.28/6.02 GB for DPTree/FASTFAIR/FPTree/WORT respectively. DPTree uses about 34% more space than FASTFAIR and FPTree because it maintains two sets of metadata. However, the space consumption mainly comes from the fingerprints and order array, which are for accelerating search operations. Moreover, crash-consistency is still maintained without them. After removing these metadata, DPTree requires only about 5% more space than FPTree. On the other hand, WORT takes much more space than other indices due to its small radix and external allocation of payloads.

In-place Merge vs. Out-of-place Merge. To show that coarse-grained versioning indeed reduces PM writes, we compare the in-place merge algorithm, which is enabled by coarse-grained versioning, with the out-of-place merge. We replace the leaf list in the original base tree with a continuous PM region where the entries are stored compactly in sorted order. We insert 5M random key-value entries and vary the entry size from 16B to 64B. The number of flushes to PM per entry is shown in Figure 8d. We notice the out-of-place merge incurs up to 5.8x more PM flushes as the entry size increases. When the entry size decreases, the out-of-place merge incurs fewer flushes as a cache line contains more entries. However, it still requires 2.1x more flushes at least. Therefore, coarse-grained versioning is effective in reducing PM writes compare to traditional out-of-place merging.

Different Leaf Search Schemes. In this experiment, we compare our hybrid linear probing with fingerprints with two other search schemes: linear-probing-only, fingerprints-only(FPTree style). We first populate DPTree with 50M entries varying from 16B to 64B and perform 50M successful/unsuccessful point lookups. The average latency of lookups is shown in Figure 10. For successful lookups, linear probing with fingerprints causes the lowest latency as the entry size increases. The fingerprints-only probing scheme performs the worst in this case as it needs to check half of the fingerprints on average. The linear-probing-only scheme performs better than the fingerprint-only scheme. This is because the correct entry is near the start of the probing position most of the time. However, the latency degrades as the entry size increases because of the increased number of cache lines probed. For unsuccessful lookups, the schemes with fingerprints stand out. This is because they mostly only check fingerprints. Therefore they show only a slight increase in latency as the entry size grows. The linear probing with fingerprints scheme reduces latency even further as it requires much fewer fingerprint checks.

4.3 PM-Optimized Buffer Managers

Traditional buffer managers utilize DRAM as a faster cache in front of the slower block-oriented storage device to improve performance. There have been works [25, 7, 14] on optimizing buffer manager for persistent memory. The 3-Tier buffer management approach proposed by Renen et al. [25] incorporates PM as an additional persistent layer between DRAM and SSD/HDD. It also introduces several key optimizations enabled by persistent memory. As DPTree contains DRAM components too, we are interested in comparing DPTree with this approach for indexing purposes.

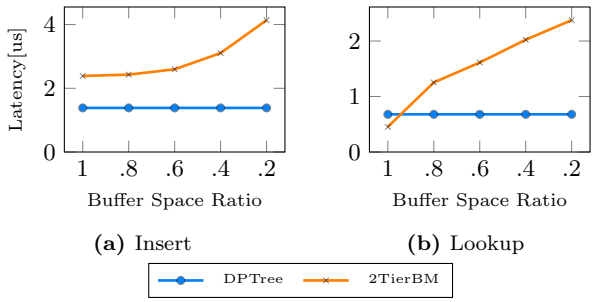


Figure 11: DPTree vs. 2-Tier-BM: 50M 16B key value pairs with uniformly distributed 8B keys

For a fair comparison, we implemented a BTree using 2-Tier (DRAM and PM) buffer manager design with key optimizations including *cache-line grained pages*, *mini pages* and *pointer swizzling*. To guarantee the crash consistency of the BTree, we implement text-book style physiological redo/undo logging on PM. We configure the 2-Tier-BM(Buffer Manager) to use 16KB pages and employ a hashing-based leaf. We perform 50M insertions followed by 50M lookups of random 8B keys (the worst case for both indices). The results are shown in Figure 11. The x-axis is the percentage of leaves in DRAM. We fix DPTree's merge threshold of R at 0.1. Therefore DPTree buffers at most 10% of the data. To keep recovery fast, we set the log capacity of 2-Tier-BM to be 30% of the entire key-value set.

We see that DPTree consistently outperforms 2-Tier-BM in insertion workload. On average, our implementation of 2-Tier-BM requires 8.5 flushes and 2 fences per insertion at 100% buffer ratio compares to the constant 2.2 flushes of DPTree. This is because, in physiological logging, every update requires logging the PM address, before and after image of the value in that address. This results in more data to be logged compare to DPTree, and incurs 4 flushes per update shown in Table 4.2. On the other hand, logging more data results in more frequent checkpoints (because log is full) which swaps out dirty pages. This results in around 4.5 flushes per op on average. As the buffer ratio decreases, latency degrades because of the increased number of expensive page swaps which results in flushes. For sorted leaf, the flush overhead is much more pronounced (up to 35X more flushes). This is because maintaining sorted order requires half of the key-value pairs to be moved and logged. For lookups, 2-Tier-BM outperforms DPTree by 50% only when it is entirely in DRAM. However, its latency degrades rapidly due to more page swaps as the buffer ratio decreases.

4.4 YCSB Workload

We then evaluate the concurrent indices using YCSB[10]. We vary the portions of index operations and generate the following workload mixtures: 1) **InsertOnly**: 100% inserts of 50M keys in total. 2) **ReadOnly:** 100% lookups. 3) **ReadUpdate:** 50% reads and 50% updates. 4) **ScanInsert:** 95% scans and 5% inserts. 5) **Mixed:** 50% lookups, 20% updates, 10% scans and 20% inserts.

The average scan selectivity is 0.001% with a standard

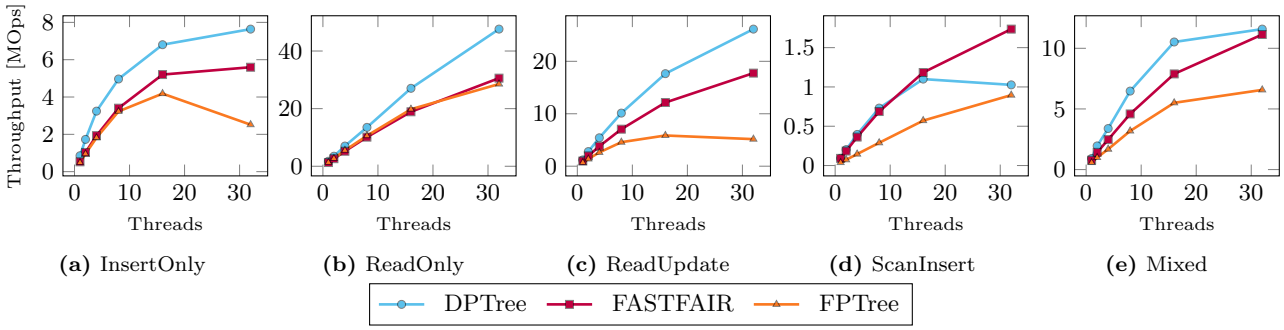


Figure 12: Throughput of YCSB Workloads with Random Key

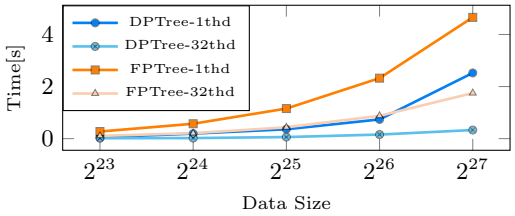


Figure 13: Recovery Performance

deviation of 0.0006%. For each workload, we first populate the index with 50M random 64-bit integer keys and then run the corresponding workload. The result of each workload reported is the average of 3 runs.

We use the default settings for FPTree and FASTFAIR described in the original papers[23, 12]. We notice that FASTFAIR[12] did not address concurrent update. Thus, we add our implementation: to perform a search first and then lock the leaf node and update the pointer in the leaf using 64-bit failure-atomic write. For concurrent DPTree, we set the number of log partition to 64, the merge threshold R to 0.07, the bloom filter error rate to 0.03, the base tree leaf node size to 8KB, the log page size to 64KB. The results are shown in Figure 12.

InsertOnly. For this workload, DPTree consistently outperforms others in terms of throughput. DPTree shows similar scalability to FASTFAIR with up to 1.36x higher throughput at 32 threads. FPTree starts with similar scalability but degrades after 20 threads due to frequent transaction aborts. Since the Optane DIMM has a write bandwidth 1/6 that of DRAM [13], all indices plateau at high thread count. However, DPTree allows more concurrency to saturate the bandwidth due to reduced PM writes per operation.

ReadOnly. For ReadOnly workload results shown in Figure 12b, DPTree stands out with 1.32-1.58x and 1.37-1.7x better lookup throughput than FASTFAIR and FPTree as thread count increases. With 32 threads, DPTree scales to 27.3x whereas FASTFAIR and FPTree scale to 23.9x and 21.6x, respectively. From the CPU statistics collected using Linux Perf tool in Table 2, DPTree has the slightest increase in CPU metrics when the number of threads increases to 32, showing the scalability of our synchronization method. The other reason that DPTree has the highest scalability is that the hash lookup inside leaf requires less PM bandwidth than the leaf scan in FPTree and FASTFAIR.

ReadUpdate. For this workload, DPTree, FASTFAIR and FPTree scale to 21.9x, 18.1x, and 7.5x with 32 threads, respectively. Figure 12c shows that DPTree has the highest throughput for all thread configurations, outperforming other indices by up to 1.7-5.07x. As shown in Table 2, DPTree has the smallest overhead across all metrics and has the smallest increase in metrics with 32 threads thanks to the efficient lookup and update (one PM write). FPTree performs the worst in this case as its update procedure requires at least three PM writes. The unsatisfactory scalability of FPTree is also reflected in the table where cycles and instruction count rises significantly when the number of threads is at 32. This is also partly attributed to the increased abort rate of TSX at high thread count.

ScanInsert and Mixed. For this workload, FASTFAIR maintains the highest scan throughput across all thread count because of the sorted leaf design. DPTree shows similar throughput to FASTFAIR and is superior throughput over FPTree by 2-3x when thread count is less than 20 thanks to the order array. However, DPTree plateaus after 20 threads because the random access inside large leaf requires more PM bandwidth. For Mixed workload, DPTree, FASTFAIR, and FPTree scale to 14x, 15.92x, and 8.2x, respectively. DPTree maintains the highest throughput among all thread count, showing the ability to scale under various access patterns.

Table 2: CPU Stats of Workloads with Random 8B Keys

Index	Cycles	Insts	L3Miss	BrMiss
ReadUpdate 1 thread/32 threads per op				
DPTree	1768/2947	456/474	3.05/3.25	2.34/2.50
FASTFAIR	2523/4436	696/703	9.07/9.16	7.77/8.29
FPTree	3581/15062	730/1624	5.25/7.93	16.22/24.13
ReadOnly 1 thread/32 threads per op				
DPTree	1461/1662	444/447	2.68/2.82	1.76/1.80
FASTFAIR	1931/2552	634/651	7.55/7.69	6.95/7.01
FPTree	1876/2615	662/781	3.59/4.26	14.96/15.56

4.5 Recovery

In this section, we evaluate the recovery performance and scalability of DPTree against other index structures. To measure the impact of data size in recovery, we vary the number of key-value pairs from 2^{23} to 2^{27}. The setting for DPTree and FPTree are the same as in section 4.4. We also measure the parallel recovery performance of DPTree and FPTree. For DPTree, the parallel recovery is simple: log partitions are read and replayed to the concurrent buffer tree in parallel. Since the FPTree paper did not mention parallel recovery algorithm, we implement our own version for FPTree in three phases: *Phase 1,* Traverse the leaf layer of FPTree to collect leaves; *Phase 2,* Spawn multiple threads to read the greatest keys out of these leaves in parallel; *Phase 3,* Reconstruct the internal nodes with one thread.

The result is depicted in Figure 13. We omitted FASTFAIR since it has instantaneous recovery. For recovery with one thread, DPTree has a much better curve than FPTree (about 1.8-5.5x less recovery time). This is because, during rebuilding, FPTree spends most of the time retrieving from

PM, the maximum key in each leaf. Whereas DPTree rebuilds the buffer tree from the PLog, which is only a small fraction of the complete key-value set. As for rebuilding internal nodes of the base tree, it takes up negligible 0.2% in the total recovery time. This is because leaf nodes in the base tree are large, and the maximum key is already computed in leaf nodes, reducing PM reads significantly. For recovery with 32 threads, DPTree can achieve up to 8x speedup over one thread, saturating the memory bandwidth of our system. FPTree, however, has only 2.6x speedup and up to 5.46x slower than DPTree. We find that with one thread, in phase (1) and (3), FPTree spends 28% of the time (each taking 26% and 2%). Phase (2) is able to scale to 11x, which saturates the PM read bandwidth of the system. However, the other two phases are hard to parallelize and turn out to be the bottlenecks.

5. RELATED WORK

DPTree is inspired by previous works on persistent indices, merge trees and concurrent KV stores.

PM-Only Persistent Indices. For PM-Only structures, data is completely in PM, enabling the possibility of near-instant recovery. Venkataraman et al [26] proposed CDDS B-Tree that uses sorted leaf and fine-grained versioning for each entry to ensure consistency and enable concurrency. Such design requires extensive `persist`s and garbage collection. To reduce `persist`s, Chen et al. [9] proposed wB$^+$-Tree that keeps leaf unsorted and relies on 8B atomic write for bitmap and logging to achieve consistency. It also employs slot-array to accelerate searches. Similar to wB$^+$-Tree, NV-Tree [31], a persistent B$^+$-Tree proposed by Yang et al., employs unsorted leaf with bitmap and reduced persistence guarantee of internal nodes. Internal nodes are rebuilt from the persistent leaves on recovery. This design requires sorting for range queries. FASTFAIR [12], a persistent and concurrent B$^+$-Tree by Hwang et al., propose FAST(Failure-Atomic ShifT) and FAIR techniques to keep node sorted by exploiting properties of modern CPUs. The BzTree [6] by Arulraj et al. employs PMwCAS (Persistent Multi-word CAS) [28] to achieve persistence and concurrency in B$^+$-Tree. However, BzTree requires at least 9 `persist`s per insert (8 from PMwCAS). The failure-atomic 8B update technique is also used to develop a persistent radix tree called WORT [15] proposed by Lee et al. However, radix tree suffers from poor range scan performance. This is even more pronounced in WORT since key-value payloads are not stored in-line. DPTree differs from these works in that it amortizes the persistence overhead of metadata, such as bitmap and slot-array, over a batch of updates.

DRAM-PM Persistent Indices. For this kind of data structures, DRAM is used for auxiliary data that is rebuilt on recovery. Since DRAM has lower latency than PM, this scheme usually results in improved performance at the cost of longer recovery time. FPTree [23], a persistent and concurrent B$^+$-Tree proposed by Oukid et al., employs such design. The internal nodes are placed in DRAM and leaves in PM. Bitmap, logging, and fingerprints are used for crash-consistency and reducing PM reads. Similar to NV-Tree, FPTree has to perform sorting for range queries. DPTree, on the other hand, uses order-array to avoid sorting. Another related work is HiKV[30] proposed by Xia et al. HiKV combines persistent hashtable and volatile B$^+$-Tree. Such design suffers from inefficient range scan and high recovery cost due to the rebuilding of the entire B$^+$-Tree.

Merge Trees. The well-known LSM-Tree[22] uses multiple tree structures to form a logical index. DPTree adopts the ideas of logging and leveling but differs from LSM-Tree in many ways. One one hand, DPTree employs only two levels to reduce read latency. The two-level design also significantly reduces the worst case write amplification in LSM-Trees. On the other hand, DPTree employs in-place merging optimized for PM instead of out-of-place merging optimized for block-based storage. Another relevant work is the *dual-stage index architecture* proposed by Zhang et al. [32]. DPTree builds on top of this work but differs in several ways. First, their work targets indexing in DRAM without considering durability. Second, their work focuses on saving space whereas DPTree focuses on optimizing index latency in such DRAM-PM hybrid systems. Lastly, their work lacks designs for concurrency which our work complements.

Concurrent Log-Structured Stores. To unleash the power of multi-core processors, researchers study concurrent and scalable log-structured KV stores. cLSM [11] increases concurrency of its memtable with reader/writer lock and concurrent skip list. However, it's scalability is limited due to the use of shared counters. Nibble [21] improves the scalability of in-memory KV store using techniques including partitioning, optimistic concurrent lookups, and multi-head logs. However, it comes at the expense of range scan ability and durability. Different from these works, DPTree addresses the scalability problem by carefully avoiding contention on shared states and introducing concurrent partitioned PM logs without sacrificing range scan and durability.

6. CONCLUSION

In this paper, we proposed DPTree, a novel index structure designed for DRAM-PM hybrid systems. DPTree employed novel techniques such as flush-optimized persistent logging, coarse-grained versioning, hash-based node design, and a crash-consistent in-place merge algorithm tailored for PM. As a result, DPTree reduced flush overhead considerably for cache-line-sized key-value payloads. The combination of Optimistic Lock Coupling B$^+$-Tree, coarse versioning, and in-place merging enabled DPTree to be concurrent and scalable in multi-core environment.

We conducted a comprehensive experimental evaluation of DPTree against state-of-the-art PM indices. We found that the single-threaded DPTree has the lowest flush overhead and highest search throughput among those evaluated. For concurrent workloads, DPTree exhibits good scalability for basic operations. In terms of recovery, DPTree achieves both better and more scalable performance than FPTree for single-threaded and parallel recovery.

As future work, we consider querying the log directly without DRAM buffer, e.g., structuring the PM log as a persistent hash table for both buffering and durability, to reduce the complexity of the design.

Acknowledgement

We would like to thank our anonymous shepherd and reviewers for their guidance and insightful comments on this work. We would also like to thank Dr. Bin Cui, Jinming Hu, and Dr. Xinyuan Luo for their useful suggestions. We would like to give special thanks to Alibaba Group for providing the

PM hardware and help for the work. This research is supported by National Basic Research Program of China(Grant No. 2015CB352402), National Natural Science Foundation of China(Grant No. 61672455), and Natural Science Foundation of Zhejiang Province(Grant No. LY18F020005).

7. REFERENCES

[1] 3d xpoint™: A breakthrough in non-volatile memory technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[2] Aerospike performance on intel optane persistent memory. http://pages.aerospike.com/rs/229-XUE-318/images/Aerospike_Solution_Brief__Intel-Optane.pdf.

[3] Intel® optane™ dc persistent memory: A major advance in memory and storage architecture. https://software.intel.com/en-us/blogs/2018/10/30/intel-optane-dc-persistent-memory-a-major-advance-in-memory-and-storage-architecture.

[4] Intel® optane™ dc persistent memory intel virtual event. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[5] r. 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). *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. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[7] J. Arulraj, A. Pavlo, and K. T. Malladi. Multi-tier buffer management and storage system design for non-volatile memory, 2019.

[8] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, 2001.

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

[10] 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, New York, NY, USA, 2010. ACM.

[11] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 32:1–32:14, New York, NY, USA, 2015. ACM.

[12] D. Hwang, W.-H. Kim, Y. Won, and B. Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 187–200, Berkeley, CA, USA, 2018. USENIX Association.

[13] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the intel optane dc persistent memory module, 2019.

[14] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 691–706, New York, NY, USA, 2015. ACM.

[15] 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 *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 257–270, Berkeley, CA, USA, 2017. USENIX Association.

[16] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, pages 3:1–3:8, New York, NY, USA, 2016. ACM.

[17] 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, Washington, DC, USA, 2013. IEEE Computer Society.

[18] J. Liu and S. Chen. Initial experience with 3d xpoint main memory. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*, pages 300–305. IEEE, 2019.

[19] 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, New York, NY, USA, 2012. ACM.

[20] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[21] A. Merritt, A. Gavrilovska, Y. Chen, and D. Milojicic. Concurrent log-structured memory for many-core key-value stores. *Proc. VLDB Endow.*, 11(4):458–471, Dec. 2017.

[22] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[23] 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, New York, NY, USA, 2016. ACM.

[24] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner. nvm malloc: Memory allocation for nvram. *ADMS@ VLDB*, 15:61–72, 2015.

[25] 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, New York, NY, USA, 2018. ACM.

[26] 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 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[27] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[28] T. Wang, J. Levandoski, and P.-A. Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472. IEEE, 2018.

[29] 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.

[30] F. Xia, D. Jiang, J. Xiong, and N. Sun. Hikv: A hybrid index key-value store for dram-pm memory systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 349–362, Berkeley, CA, USA, 2017. USENIX Association.

[31] 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, Berkeley, CA, USA, 2015. USENIX Association.

[32] 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, New York, NY, USA, 2016. ACM.