# Building Scalable NVM-based B+tree with HTM

Mengxing Liu*
liu-mx15@mails.tsinghua.edu.cn
Tsinghua University

Jiankai Xing*
xjk17@mails.tsinghua.edu.cn
Tsinghua University

Kang Chen *†
chenkang@tsinghua.edu.cn
Tsinghua University

Yongwei Wu *
wuyw@tsinghua.edu.cn
Tsinghua University

## ABSTRACT

Emerging on-volatile memory (NVM) opens an opportunity to build durable data structures. However, to build a highly efficient complex data structure like B+tree on NVM is not easy. We investigate the essential performance bottleneck for NVM-based B+tree. Even with a single-core CPU, the performance is limited by the **atomic-write size** which plays an essential role in the trade-off between the persistent overhead and keeping leaf node entries sorted. For the multi-core setting, the **overlapping of concurrency and persistency** is key to the system scalability.

Based on the analysis, we propose RNTree, a durable NVM-based B+tree using the hardware transactional memory (HTM). Our way of using HTM can actually address both problems mentioned above simultaneously. (1) HTM can use cache-line granularity to provide larger atomic-write size. Based on this, we propose a new slot-array approach which traces the order of entries in the leaf nodes while still reducing the number of persistent instructions. (2) With careful design, RNTree moves slow persistent instructions out of critical sections and proposes the dual slot array design, to extract more concurrency. For single thread, RNTree achieves 1.44×/4.2× higher throughput for single-key operations and range queries respectively. For multiple threads, the throughput of RNTree is 2.3× higher than state-of-the-art works.

**ACM Reference Format:**
Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu . 2019. Building Scalable NVM-based B+tree with HTM . In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337827

## 1 INTRODUCTION

Emerging non-volatile memory (NVM) technologies [1] are promising because they offer non-volatility, byte-addressability and fast access at the same time. Empirical studies [2–7] suggest that NVM should be attached to the memory subsystem directly, accessed by normal load/store instructions to avoid the overhead of the legacy block-oriented file systems and data serialization/deserialization.

This paper focuses on building non-volatile B+tree which represents an important class of tree-based sorted structures. B+tree can support `find`, `range query` as *read only* operations and `insert`, `remove`, `update` as *modify* operations. Compared to other range indexes, e.g., skip list, B+tree has smaller tree depth, better cache locality and dense data layout. Thus, even B+tree was firstly introduced for storage systems on disks, there are many works [6–9] on B+tree for the NVM.

However, even the memory is non-volatile, data in the cache hierarchy above the memory are transient. Transient data in CPU caches can be persisted by either *explicit cache line flush instructions*, e.g., `CLFLUSH`, `CLWB`, or *implicit cache eviction*. Implicit cache eviction is uncontrollable. The carefully ordered memory instructions in user programs might be reordered. Programmers usually enforce the order by a cache line flush instruction followed by a memory fence instruction, e.g., `MEMFENCE`. We can call such instruction compound as one *persistent instruction* as it flushes the data from the cache to the NVM, prevents any re-order and waits until the data are physically persisted. The persistent instruction incurs non-trivial overhead. Most researches on NVM focus on how to reduce persistent instructions as much as possible.

Traditional structure of B+tree does not make the best use of the byte-addressability of NVM. A classic B+tree has large nodes (usually several kilos of bytes in each node) and requires items in each node to be sorted for fast find and range query. Any insertion or deletion of an item in the node needs the rewrite of the whole node. It brings the problem of write amplification that *one modification of the data structure needs multiple writes*. On disk, it is not a problem as long as the size of the node does not exceed the size of one disk block. On the contrary, due to the byte-addressability of the NVM, write amplification incurs unnecessary performance penalties. Some current works [6, 8, 9] try to solve this problem by using append-only strategy. For example, NVTree [8] appends a log entry at the end of the leaf node without sorting in each `modify` operation. But *read-only* operations have to scan the whole nodes. Other work [7] tries to keep the order in the leaf node but requiring more persistent instructions. **There is a trade-off between the sorted leaf node and the persistent write overhead. (Challenge 1.)**

*Persistent order* also makes concurrent programming complicated. It is hard to guarantee the crash consistency and the correct concurrent access (linearizability) at the same time. Most of the current works [7–10] only support single thread. FPTree [6] uses

*Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China
†Corresponding author: Kang Chen (chenkang@tsinghua.edu.cn)


*ICPP 2019, August 5–8, 2019, Kyoto, Japan*

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337827

**Table 1: Overview of RNTree vs. Recent Works.**

| | Writes | Sorted | Concurrency |
|---|---|---|---|
| CDDS Tree | $L^*$ | √ | × |
| NV-Tree | 2 | × | × |
| wBtree | 4 | √ | × |
| FPtree | 3 | × | Coarse grained |
| RNTree | 2 | √ | Fine grained |

the *selective concurrency* , i.e., the whole leaf node is locked when applying a `modify` operation. Essentially, FPTree creates a critical section to guarantee linearizability at first, and then use persistent instructions to guarantee the crash consistency in the critical section. This *decoupled* design is easy to implement, but far from effective. When a `modify` operation is applied, other operations, including *read only* operations, are blocked. **How to reduce the synchronization overhead is critical to improve the scalability. (Challenge 2.)**

To address the above challenges, this paper proposes RNTree (**R**TM **N**VM B+tree), a high efficient and scalable NVM B+tree implementation. The key to achieve high performance is through hardware transactional memory (HTM) elegantly and carefully.

**Challenge 1:** To avoid the unsorted leaf node and persistent write overhead, HTM is used because it has a much larger *atomic-write size* than operands in normal instructions. The atomic-write size represents the granularity that the CPU can guarantee the atomicity when writing to underlying memory, usually at most 8 bytes. HTM guarantees that store operations within a single transaction are not written into memory sub-system until the transaction commits successfully. Thus, we can increase the atomic-write size from 8 bytes to a cache-line size (typically 64 bytes). HTM can help to increase the atomic-write size as in previous works [11, 12]. With the increased atomic-write size, it gets more chances to capture the order information of entries in the leaf node. Meanwhile, two persistent instructions are enough in a *modify* operation in RNTree, one for the raw data and the other for the metadata while still keeping the entries sorted. Using HTM for increasing the *atomic-write size* can effectively solve the trade-off.

**Challenge 2:** To reduce the potential synchronization overhead, we propose an *overlapping* design for concurrency control. Investigating the implementation details, a B+tree `modify` operation consists of several steps after it reaches the corresponding leaf node (regardless of different implementations): (1) *Allocate* a log entry; (2) *Write* data in the entry; (3) *Flush* the data; (4) *Update the metadata*. RNTree does not try to execute all these steps in one critical section. Instead, it overlaps the concurrency control and the crash consistency, so that slow *flush* step does not block others' execution. Meanwhile, we propose the *dual slot array design* to reduce the read-writer contention.

We summarize the main differences between RNTree and previous works [6–8, 10] in Table 1. *Writes* denotes the number of persistent instructions needed for each modification. $L^*$ is the number of entries in the leaf node. *Sorted* denotes whether to keep the leaf node sorted or not. Concurrency denotes whether or how to support concurrent accesses. RNTree achieves the least number of persistent instructions and keeping entries sorted in each leaf node. RNTree has better scalability than existing systems.

The main contributions are as follows.

- **Investigate the essential performance bottleneck for the NVM-based B+tree.** Small atomic-write size incurs the trade-off between the higher persistent overhead and sorted leaf nodes. Meanwhile, persistent instructions can bring severe overheads, and naive implementation of concurrency would make slow persistent instructions hurt the scalability.
- **Solve the trade-off between the persistent overhead and sorted leaf nodes.** RNTree uses HTM to increase the atomic-write size from 8 bytes to 64 bytes. A cache-line size metadata is used to trace the order of entries in leaf nodes. The number of persistent instructions is minimum for `modify` operations while still keeping leaf nodes sorted.
- **Extract more concurrency with overlapping the concurrent and persistent algorithm, and propose the new dual slot array design**. RNTree overlaps the design of persistency and concurrency, excludes the slow *persistent* step out of the critical section. Meanwhile, RNTree adopts a more friendly technique for read-only operations, while still guarantees the strictest consistent model, i.e., linearizability.

## 2 BACKGROUND

### 2.1 NVM Programming Model

The discussion about NVM technologies can be found in [1] as well as how to benefit existing applications [7–10, 13]. We follows the NVM.PM.FILE mode described by SNIA[1]. In this mode, NVM devices are managed by a PM-aware (persistent memory aware) file system [12, 14, 15], which can support direct access (DAX) with *mmap* system call, providing direct access to the NVM through load/store instructions. Applications should maintain the durability and consistency of data. However, the carefully designed order in applications can be easily broken by out-of-order cache evictions in modern CPUs. To avoid unexpected eviction, one need to use persistent instructions (`CLFLUSH`, `MEMFENCE`) to enforce the write order. Though the access latency of NVM is much smaller than disk or SSD, it still incurs up to 100 ns latency [1]. We need to reduce the number of persistent instructions to boost the performance.

Atomic-write size is the size that CPUs can guarantee the atomicity when writing to the NVM. It mainly determines the number of persistent instructions required. Additional persistent orders are mainly caused by the small atomic-write size, e.g., current x86 CPUs only support 8 bytes as the atomic-write size. Though the flush can occur in cache-line size (64bytes), when to flush is uncontrollable. Previous works use undo logs or redo logs [2, 3, 5, 16], which require careful design and additional orders to achieve atomicity.

### 2.2 Hardware Transactional Memory

Hardware Transactional Memory (HTM) is a tool that simplifies concurrency programming by allowing a batch of load and store instructions atomically visible. There are several commercially available HTM implementations such as Intel Restricted Transactional

[1] https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf

Memory (RTM). We use RTM in this work, but our algorithms are valid for other HTM implementations.

There are several limitations when using HTM and NVM simultaneously. At run-time, RTM buffers the modified data in L1 cache before commits. Thus, one of the major limitations is that a transaction will abort if the working set size (i.e., data read and write in the transaction) exceeds the L1 cache size [13]. Another limitation is that cache-line flush instructions (e.g. `CLFLUSH`, `CLWB`) inside a transaction will always abort the transaction. Therefore, a normal mutex lock can be used to replace HTM when executing cache-line flush instructions to flush data from the cache to NVM.

The most required characteristic of HTM for our work is that HTM can increase the write-atomic size to 64 bytes (size of a cache-line). HTM guarantees that a dirty cache-line incurred by a store instruction remains in the cache, without flushing to the memory. If the system crashes before the transaction finishes, the dirty data will be lost without hurting the consistency of the NVM.

# 3 PERFORMACE ANAYSIS

In this section, we introduce several related works and discuss trade-offs of designing the persistent B+tree from different perspectives.

## 3.1 Existing Persistent B+tree

**CDDS B-Tree** (Consistent and Durable Data Structure B-Tree) [10] is a multi-version B-tree. When a tree node is updated, a new copy will be created and tagged with a new version without overwriting the original entry for recoverability and consistency. CDDS B-tree maintains entries sorted and suffers from the *write amplification*.

**NVTree** [8] reduces the expensive persistent instructions by employing an append-only strategy. When a new entry is inserted, the data are appended at the end of the leaf node without keeping entries sorted. Only two persistent instructions are needed, one for the entry and the other for the count of entries. Without sorted leaf nodes, `find` and range query operations have to scan all logs.

**wB+Tree** Compared to NVTree, wB+Tree [7] proposes to sort the entries via the slot array holding the actual positions of entries. An additional *valid bit* is used to identify the states (valid or not) of the slot array if the slot array is larger than 8 bytes and cannot be updated atomically. Thus, wB+Tree requires another two persistent instructions to set/reset the valid bit for each `modify` operation, leading to higher write overhead.

**FPTree** Similar to NVTree, FPTree [6] employs an append-only strategy for leaf nodes. One-byte key hash is used to reduce the cache miss in `find`, although leaf nodes are still unsorted. FPTree uses HTM to support concurrent accesses. Traversing from the tree root to the leaf node is wrapped by HTM. After reaching the leaf node, an explicit mutex lock is acquired for the whole leaf node operation. According to our investigation, operations on the leaf node take more than 70% time in a request. Thus, locking the whole leaf node would incur high scalability penalties.

In the following, we start with the analysis of different design trade-offs before making the design decisions for our RNTree.

## 3.2 Write Amplification vs. Sorted Leaf Node

The primary challenge in the NVM programming is caused by high overhead of persistent instructions. Write amplification should be avoided as much as possible. The extra persistent instructions are essentially caused by the small atomic-write size in instructions. If the data size is larger than the atomic-write size, it is unavoidable to split the operation into several steps. As system could crash at any point of the execution, there must be at least one extra *state* variable indicating the running stage before the crash. The wB+tree takes this approach, using a *valid bit* to mark the validity of the slot array, which needs 4 persistent instructions for each *modify* operation. For example, each `insert` operation needs to: (1) append a log at the end of the leaf node, and flush the log; (2) set the valid bit as false and flush the valid bit;(3) insert a new slot and keep the slot array sorted, flush the new slot array; (4) set the valid bit as true and flush the valid bit. After the system crash, if the valid bit is true, wB+tree uses the slot array directly. Otherwise, wB+tree uses key value (In the rest of the paper, we will use acronym KV for representing key-value pairs) entries in logs to recover the slot array. This strategy keeps crash consistency for wB+tree, but expensive.

On the contrary, other works relax the property of *sorted leaf node* to reduce the write amplification. Instead, they apply the append-only strategy. Such design writes and flushes logs first, and then atomically modifies and flushes the metadata whose size is limited to 8 bytes. The access to the data is guided by the metadata. The old and new versions of metadata both represent consistent states. Thus no extra logging and shadowing are required. For example, in the NVTree, the metadata is the log length counter. In the FPTree, the metadata is the bitmap, in which each bit represents a used log entry. It is evident that such small atomic write size cannot capture the order of entries in a leaf node. Thus, we need to solve **the trade-off between the persistent overhead and sorted leaf nodes.**

## 3.3 Conditional Write vs. Append-only

The widely adopted append-only strategy introduced before is not friendly to support *conditional write*. Conditional write means an `insert` succeeds only if there is no data record with the same key, while an `update` or `delete` operation succeeds only if there is a record that has the same key. This property is critical if B+tree serves as primary keys or the *unique constraint* is applied. *Conditional write* is the core feature for relational databases and many key value stores[17, 18].

To support the conditional write, NVTree and FPTree have to scan all logs in the leaf node to check the existence of a key. Our experiments show that the unsorted leaf node incurs about 19% overhead to provide conditional write for `modify` operations. We should consider **how to support fast conditional write**.

## 3.4 Easy Programming vs. High Scalability

Multi-threading is the key to saturate the NVM hardware, but none of the previous works does it sufficiently well. Most of the current works [7, 8, 10] do not support multi-threading because of its complexity. FPTree [6] can support concurrent accesses by locking the whole leaf node before modifications. Logs and metadata are flushed inside critical sections protected by locks. Such decoupling approach makes the concurrent programming easier but potentially limits the scalability. Coarse-grained lock behaves worse in NVM than in DRAM because persistent instructions consume order-of-magnitudes more CPU cycles than normal instructions. Putting

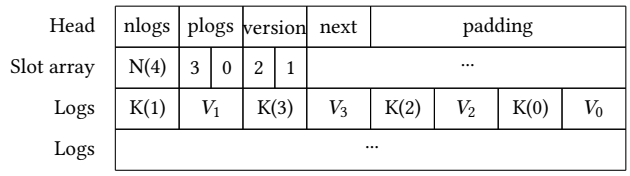


**Figure 1: Data Layout of Leaf Nodes.**

persistent instructions inside a critical section will significantly prolong the time inside the critical section, possibly leading to higher contention. According to our evaluation, the throughput of FPTree drops an order of magnitude slower under skewed workloads. Thus, even more programming efforts are required, we need to consider **reducing the influence of persistent instructions on multi-threading.**

### 3.5 Strict consistency vs. More concurrency

Consistent models for concurrent objects on NVM are different from their counterparts on DRAM. Programmers have to take two aspects under considerations simultaneously: *persistent order* for the crash consistency, and *logic order* for the correct concurrent control of multiple threads. The former makes sure that the data structure can be recovered to a consistent state at any point of the execution after crash. The later makes sure that concurrent accesses do not break the data structure and should return *correct* results, e.g., mostly linearizability [19]. Data structures are said to have *durable linearizability* [20] if any concurrent execution is similar to a sequential execution even with system crashes.

The difficulty of concurrent programming of NVM is that crash consistency and linearizability are not composable. If an algorithm is linearizable (not considering crashes), and is crash consistent for a single thread, it might **not** have durable linearizability. The NVTree is such a case. When an item is inserted (updated, or deleted) in the NVTree, a log entry is appended. The counter of log entries denoted as *nElement* is increased by an atomic add instruction. `Find` operation would scan all log entries in the leaf node within the boundary of *nElement* to find the exact key, without blocking any ongoing modifications. However, the draft algorithm in the paper can incur a *read-uncommitted anomaly*. A write thread increases the *nElement* but without flushing, and a concurrent read thread reads the new *nElement* and returns the result based on the new *nElement*. Suppose at this point the system crashes. After recovery, *nElement* is stale as if the writer thread has not executed. In this case, the read thread returns a value that does not exist in the tree, breaking the linearizability. In some weaker consistent models, this read-uncommitted anomaly is allowed for better performance. But the durable linearizability is the most desirable consistency model for NVM and easiest for programmers to write correct codes. FPTree and RNTree both have durable linearizability. Thus, **it is critical to extract more concurrency with strict consistency model**.

## 4 RNTREE DESIGN

RNTree tries to deal with the challenges analysed before. Specifically, (1) RNTree solves the trade-off between persistent overhead and sorted leaf node, which also solve the conditional write problem. (2) RNTree overlaps persistency and concurrency, moving the persistent instructions outside critical sections, effectively reducing the contention. (3) RNTree supports durable linearizability, i.e., the strictest consistency model. In this section, we present the design of RNTree to achieve these goals.

Similar to other NVM based B+tree [6, 8], we store all leaf nodes in NVM, and all internal nodes in volatile DRAM. Storing internal nodes in DRAM can reduce the overhead of tree re-balancing, and is friendly to HTM because cache line flush instructions are omitted. If system crashes or reboots, internal nodes can be reconstructed from leaf nodes.

### 4.1 Cache-Line Size Slot Array

RNTree uses the cache-line size slot array to increase the atomic write size for the metadata. An `insert` or an `update` operation appends a log at the end of the KV area for the leaf node. An indirect slot array remembers the sorted order of KV entries. Figure 1 shows the data structure of a leaf node. Each row represents a cache line. The first cache-line stores auxiliary data, including *nlogs* (the number of logs allocated but may not be persisted), *plogs* (the number of persisted logs), *version* ( the leaf node version ) and *next* (the pointer to the next leaf node). The second cache-line stores the slot array. The first byte stores the length of the slot array (4 in the figure). The rest 63 bytes record the order of log entries. For example, the smallest key is stored in *Log*[3]. Log entries start from the third cache line, and they are aligned to cache-line size. The indirect slot array can support binary search for `find` or range query operations. Note that the slot array and log entries are crash consistent. Variables like *nlogs* and *plogs* are not. But they can be recovered from the surviving data in the NVM.

If all modifications on the slot array are wrapped in an HTM transaction, the slot array can be flushed to NVM atomically after the transaction commit. Step (2) and step (4) in wB+tree (described in Section 3.2) can be eliminated. In the case of system crashes, the slot array persisted in NVM is either in the old state or in the new state. The leaf node is always in a consistent state. Thus, only **2** persistent instructions are required.

### 4.2 Overlapping Persistency and Concurrency

A `modify` operation needs four steps: (1) *allocate* a log entry; (2) *write data* in the log entry; (3) *flush* the log entry; (4) *update the metadata*. We test the CPU cycles consumed by all steps and find that the *flush* step consumes most CPU cycles in a *modify* operation. Previous works use the *decoupling* design that includes all steps in one critical section. Such design is easy to implement, but brings a considerable overhead. Long time spent in critical sections hurts the concurrent performance, especially for skewed workloads.

We investigate the four steps from the view of persistency and concurrency. (1) The *log allocation* step requires concurrency control; (2) The *data write* step is an ordinary code block and requires neither concurrency control nor persistency; (3) The *log flush* step requires persistency; (4) The *metadata update* step requires concurrency control.

As we can see, different steps have different requirements and we can apply an overlapping design for persistency and concurrency.

Only the first step and the last step require concurrency control. We use a compare-and-swap (CAS) instruction to allocate correct log entry for each thread, and a spin-lock to protect the update of metadata. Mutiple thread can flush logs in parallel. By overlapping the code for persistency and concurrency, the costly log flushing is moved out of the critical section and never blocks other threads.

### 4.3 Dual Slot Array

Because of *read-uncommitted anomaly* explained in Section 3.5, reader-writer contention cannot be solve easily. Readers can not see the modified slot array before it is flushed. There are two naive ways to deal with this problem: (1) Lock based: readers and writers use the same lock; (2) Version based: Writers increase the version number after each update. Readers check the version of the leaf node before and after the read. The reader will retry if the version has changed. None of the above methods is efficient.

We propose a dual slot array design. There are two slot arrays in a leaf node, one is transient, and the other is persistent. A writer will firstly persist the KV (*step*1), then modify and flush the persistent slot array (*step*2), and finally update the transient slot array (*step*3). Readers would use the transient slot array to read data. Only when a reader reads the transient slot array after *step*3, it will get the new data. The new data must be persisted here. Otherwise the reader will get the old data. Thus, the transient slot array always represents persistent data, avoiding the read-uncommitted anomaly. Dual slot array makes it possible to execute read and modify operations concurrently.

## 5 IMPLEMENTATION

### 5.1 Synchronization Building Blocks

RNTree implementation uses HTM for concurrent control whenever possible. Spin lock is still needed because persistent instructions will abort hardware transactions. Special data structures are designed for supporting synchronization. Similar to Masstree [21], an integer value is used to represent both version and lock, as in Figure 2. The *lock* bit is used by modify operations, and *splitting* bit is set when the leaf node is being split. We have built several helper functions for synchronization: the function *lock* uses a CAS instruction to set the *lock* bit, and the function *unlock* reset the *lock* bit. The function *setSplit* and *unsetSplit* sets/resets *splitting* bit respectively. The version number is increased when the splitting is finished. The function *stableVersion* returns a stable version, i.e., returns the version number when the leaf node is not splitting.

RNTree provides several low-level functions wrapped by HTM in Table 2. These functions can be considered as atomic instructions, and are the *building blocks* for implementing operations for data structures. Intel threading building blocks (TBB) library is used to implement these functions.

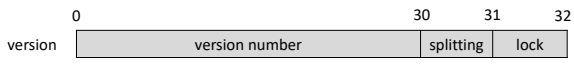


**Figure 2: Layout of the Leaf Version.**

### 5.2 Basic operations

*5.2.1 Insert.* Algorithm 1 shows the `insert` operation using three steps. (1) It directly traverses the tree to find the leaf node by `htmTreeTraverse` (line 2). (2) After reaching the leaf node, a lock-free algorithm is used to allocate a blank entry (line 3). Algorithm 2 shows how to allocate a log entry. It returns NULL if there is no log entry available. Specifically, the variable *nlogs* points to the next entry to be allocated. A CAS instruction is used to atomically increase the *nlogs*. Each thread will get its specific entry slot without interfering each other. It is possible that there is no free log entry in the current thread because the leaf node is full and being split in another thread. Current `insert` operation has to restart the traverse from the root node again (line 5), hopes that the split procedure completes then. (3) After the KV is persisted, the slot array is updated inside an HTM transaction and keeps entries ordered. Because the cache-line flush is atomic, the slot array is always consistent in NVM regardless when the system crashes.

**Algorithm 1** Insert(K key, V value)

```
1:  while True do
2:      leaf = htmTreeTraverse(key);
3:      entry = allocateEntry(leaf);
4:      if entry == NULL then
5:          continue;
6:      end if
7:      leaf.KV[entry] = (key, value);
8:      Persist(leaf.KV[entry]);
9:      lock(leaf);
10:     htmLeafUpdate(leaf, key, entry, false);
11:     Persist(leaf.slot);
12:     htmLeafCopySlot(leaf);
13:     leaf.plogs++;
14:     if leaf.plogs == LeafNodeCapacity-1 then
15:         Split(leaf);
16:     end if
17:     unlock(leaf);
18:     return;
19: end while
```

**Algorithm 2** allocateEntry(LeafNode leaf)

```
1: entry = leaf.nlogs;
2: if entry >= LeafNodeCapacity then return NULL;
3: end if
4: while CAS(&leaf.nlogs, entry, entry+1) == false do
5:     entry = leaf.nlogs;
6:     if entry >= LeafNodeCapacity then return NULL;
7:     end if
8: end while
9: return entry;
```

RNTree always uses the slot array as the source-of-truth. Line 10 in Algorithm 1 is the atomic *turning* point. Before this point, the slot array contains the old data. After this point, the new slot array is persisted. Because old log entries are not overwritten, the slot array will never point to invalid entries. *htmLeafCopySlot* is used to copy the persistent slot array to the transient slot array (line 12 in Algorithm 1).

**Table 2: HTM Functions**

| Function | Description |
|---|---|
| (Leaf) htmTreeTraverse(K) | Traverses from the root node and returns the leaf node **whose range covers the key**. |
| (bool) htmLeafUpdate(leaf, K, entry, exist) | Updates the slot array in the leaf node. The parameter *K* denotes the key, *entry* denotes the location of the KV entry, *exist* denotes whether the key should exist in the leaf node, supporting conditional write. |
| void htmLeafCopySlot(leaf) | Copy the transient slot array from the persistent slot array. |
| (int, SlotArray) htmLeafSnapshot(leaf) | Takes a snapshot of the slot array of the current leaf node. |
| void htmTreeUpdate(oldLeaf, newLeaf, separator) | Updates internal nodes, inserts a new leaf node after the old leaf node, separated by the *separator*. |

At the end of `insert` operation, the full leaf node may trigger `split` (Algorithm 3). In case of system crashes during splitting, we first log the whole leaf node in a pre-defined thread-local storage (undo logs). Then a new leaf node is allocated. All key value pairs are divided into the old leaf node and the new leaf node. In the end, an HTM function `htmTreeUpdate` is used to update the parent internal node. The version number is increased after each split.

**Algorithm 3** Split(LeafNode leaf)

```
1:  setSplit(leaf);
2:  logLeaf = leaf; Persist(logLeaf);
3:  newLeaf = NewLeafNode();
4:  split = leaf.slot[0]/2;
5:  splitKey = leaf.kv[split].key;
6:  newLeaf.slot[0] = leaf.slot[0] = split;
7:  for i in [0, 1, ..split − 1] do
8:      leaf.slot[i+1] = newLeaf.slot[i+1] = i;
9:      leaf.kv[i] = logLeaf.kv[leaf.slot[i+1]];
10:     newLeaf.kv[i] = logLeaf.kv[leaf.slot[i+1+split]];
11: end for
12: newLeaf.next = leaf.next;
13: leaf.next = newLeaf;
14: Persist(newLeaf); Persist(leaf);
15: unsetSplit(leaf);
16: htmTreeUpdate(leaf, newLeaf, splitKey);
17: free(logLeaf);
```

*5.2.2 Find.* The `find` operation is in Algorithm 4. After reaching the corresponding leaf node, a snapshot of the transient slot array is taken. Then, the binary search is applied on the snapshot. We intentionally avoid the binary search inside the HTM section to reduce the size of read-set for one HTM transaction, so as to reduce the possibility of transaction abort. The version numbers are fetched before and after `find`. If versions do not match, `find` needs retry. Unlike FPTree, RNTree rarely traverses again from the root node, unless the current leaf node has been splitted. FPTree has to abort and retry if the current leaf node is updating, which is more frequent than splitting. RNTree gains the performance benefit of `find` from the dual slot array. The dual slot array design makes the increasing of the version number from *each modification* to *each split*, effectively reduce the chance of retry. If the leaf node is not being splitted, `find` can be considered as non-blocking.

**Algorithm 4** Find(K key)

```
1: while True do
2:     leaf = htmTreeTraverse(key);
3:     v = StableVersion(leaf);
4:     N, slotArray = htmLeafSnapshot(leaf);
5:     res = binarySearch(leaf.kv, slotArray, N);
6:     if StableVersion(leaf) != v then
7:         continue;
8:     end if
9: end while
```

*5.2.3 Update and Remove.* `Update` is similar to `insert`. The updated entry will be appended to the end of the log area in the leaf node. The slot array is modified accordingly. Since obsolete entries are still kept in the log area, the number of active KV pairs (indexed by the slot array) may be smaller than the log entries allocated. The obsolete entries are recycled during `split`. If the active entries are less than half of the capacity of a leaf node, a special purpose `split` is invoked to replace the old leaf node with a new one (not displayed in Algorithm 3).

Algorithm 1 can also be used to describe `update`. If conditional write is not supported, `insert` and `update` are the same. If conditional write is supported, the only difference is in last parameter for the function *htmLeafUpdate*, *false* for `insert` and *true* for `update`.

The `remove` operation is simpler than `update`. `Remove` operation only needs to update the slot array in the leaf node.

*5.2.4 Range query.* The implementation of range query is straight forward. With provided start key and a filter function, the start item will be firstly located and the scan can proceed following *next* pointers until the filter function returns true. It is not clear how to implement range query in the existing works with unsorted leaf nodes[6–8]. A straightforward way is to sort each encountered leaf node. But sorting incurs several times higher latency.

## 5.3 Coordination Analysis

In this section, we discuss the *correctness* of RNTree under the multi-threading setting.

*5.3.1 writer-writer coordination.* Essentially, RNTree can be considered as a persistent linked list, i.e., leaf nodes, and a volatile N-ary tree, i.e., internal nodes. Leaf nodes can also be divided into log area, which contains the real KV data, and other auxiliary data,

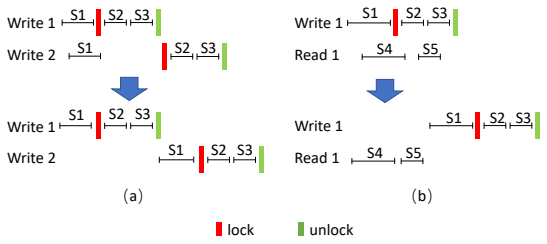


**Figure 3: In each sub-graph, the top half denotes a possible concurrent execution history, the below part denotes its corresponding linearizable history. The red and green blocks denote lock acquiring and releasing respectively, while a line denotes a sub-procedure. S1: write & flush the KV entry; S2: write & flush slot array; S3: update transient slot array; S4: read transient slot array; S5: read KV entries.**

including the slot array for keeping the sorted leaf node property. Writers in RNTree coordinate through several techniques for each part of data respectively: Any modification to internal nodes are protected by HTM functions. Any modification to the leaf node has to acquire the lock except the log entry allocation. A CAS instruction is used, instead. Thus, log flushes can run in parallel.

Figure 3(a) gives an example of two concurrent writers and how they can be linearized [19]. In this case, writer 1 and write 2 get their log entries, write and flush data simultaneously. Then they compete for the spin lock of the leaf node, and modify the structure of the leaf node one by one according the order they get the lock. The concurrent execution history is equal to a a sequential execution history indicated in the below half in Figure 3(a). That is, writers are linearized by the order they get the lock, no matter when they write and flush log entries.

Note that even updating slot arrays in leaf nodes is wrapped by an HTM function, it is protected by the spin lock from other concurrent writers. The HTM function here is for the cache-line size atomicity.

*5.3.2 reader-writer coordination.* Readers and writers may conflict in two cases: (1) one thread is reading the transient slot array when another thread is trying to update it. We solve this conflict by HTM. (2) After a reader gets the correct slot array and read logs according to it, another thread splits the leaf node and modifies log entries[2]. We use a classical optimistic concurrency control mechanism for this kind of reader-writer coordination. Every reader will get two version numbers before and after its computation respectively. If the versions are not matched, the reader has to retry. The version changes only when splitting. Thus, readers do not have to retry even with simultaneous writers, as long as there is no splitting.

It is oblivious that readers never see *un-committed* data due to the dual slot array design, but they may fail to read *committed* data. It is still linearizable [19]. Figure 3 gives an example. The writer updates and flushes the persistent slot array, but it has not updated the transient slot array. The reader will use the old transient slot array and can not see the data from the writer. In this case, *logically*, the reader happens before the writer. As a conclusion, a reader is linearized before a writer if it does not read the transient slot array the writer is about to update, even though the writer has already flushed the KV entry.

[2]Note that an ordinary update (without splitting) would not conflict with concurrent readers, because it only allocates a new log entry and never overwrites old log entries, shown in algorithm 2.

## 5.4 Recovery

All internal nodes are sorted and stored in DRAM, that they are lost after crash. `Recovery` reconstructs the internal nodes from the persisted leaf nodes stored in NVM. The `recovery` operation scans all leaf nodes, resets their *lock, nlogs, plogs* values and retrieves the greatest key in each leaf node to rebuild the whole tree. If some nodes need to `split`, e.g.crash during `split`, the recovery helps to complete the `split`. The pointer to the left-most leaf node is stored in a well-known static address for starting the `recovery`.

# 6 EVALUATION

As previous works are not open-sourced, for evaluation, we re-implement these works as faithfully as possible[3]. The structures for all the internal nodes are the same in all implementations. The only difference is the design of the leaf node.

(1) For NVTree, we abandon its static internal nodes architecture, which leads to rebuilding the whole tree and higher write overhead. We also optimize `update` for NVTree. NVTree appends a *remove* log and an *insert* log for an `update` operation. We omit the *remove* log to reduce memory flushes. And during read, we scan the log area from back to front. This strategy has the same semantic, but reduces half of the memory writes.
(2) There are two versions of wB+tree: (1) **wB+tree-SO** the size of the slot array is just 8 Bytes, no more than write-atomic size. Thus, extra persistent instructions introduced in Section 3.2 can be omitted. But it can only store 7 KV entries in each leaf nodes, leading to long tree depth and frequent tree re-balancing. (2) **wB+tree** the size of the slot array is 64 Bytes, exceeding the write-atomic size, requiring extra persistent instructions for the *valid bit*.
(3) FPTree have to support conditional write because it uses a bitmap to identify occupied logs, and log entries can be reused. FPTree cannot distinguish two logs with the same key. Thus, it must support conditional write to avoid this case, i.e., there are no two same keys in the FPTree. On the contrary, the basic implementation of NVTree does not support conditional write. If it has two logs with the same key, it chooses the latest log, as new logs always append after old logs.

FPTree and RNTree support multi-threading, while NVTree and wB+tree only support single-threading.

## 6.1 Experimental Setup

Our evaluation consists two parts. (1) Single thread evaluation compares performance of basic operations: `insert`, `find`, `update`, `remove` and range query. We investigate how large atomic-write size used can satisfy all operations. (2) Multiple threads evaluation

[3]Our codes are available at https://github.com/liumx10/ICPP-RNTree

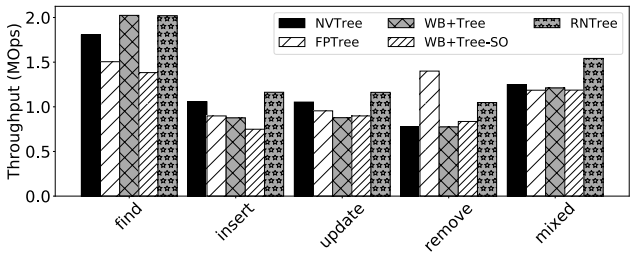


**Figure 4: Throughput of Find/Insert/Update/Remove and Mixed Operations on Single Thread.**

runs the test with various number of threads on well-known YCSB benchmarks for evaluating scalability.

**Evaluation Setup.** All evaluations use Dell PowerEdge R740 with two Intel Xeon Gold 6126 processors supporting TSX and `CLWB`, with 12 16GB NVDIMM-N chips. These NVDIMM-Ns are grouped by two 6-way interleave sets, i.e., the capacity of each set is 96 GB. The read/write lantency of the interleaved NVM are 84/140 ns, and the read/write bandwidth are 61/34 GB/s respectively. The persistent memory is managed by ext4 file system with the DAX flag and mapped into a pre-designated memory address of the application program. The results are the average of 10 runs.

## 6.2 Single-thread benchmarks

This evaluation firstly warms up the tree with 16 million KVs. Each leaf node contains at most 64 (7 in wB+tree-SO) KVs. We have tried other size of leaf nodes, but the size of 64 performs the best in general. Each operation runs for 5 seconds except `remove`. `Remove` operation runs for 100ms. Otherwise, all nodes in the tree will be removed. There is also another mixed operation benchmark in which each operation takes the same proportion. Figure 4 shows the results.

*6.2.1 Find.* RNTree and wB+tree perform the best in `find`. NVTree and FPTree suffer from slow linear search. RNTree can run 12% faster than NVTree. The performance improvement comes from the leaf nodes since all internal nodes are sorted. FPTree is even slower than NVTree because it has to calculate hash code when searching. wB+tree-SO is slow because it has at most 7 KV entries in each leaf node and can not benefit from the binary search. The small leaf node capacity also leads to longer tree depth, which means slow tree traversal.

*6.2.2 Insert.* wB+tree/FPTree/NVTree have 4/3/2 persistent instructions respectively in the `insert` operation. The results in Figure 4 show that the performance is consistent with the number of persistent instructions. NVTree is a little slower than RNTree because of its slow splitting. NVTree has to sort all data in the node before splitting. wB+tree-SO also has 2 persistent instructions for each `insert`, but its performance is the worst. The reason is its small leaf node capacity leading to more frequent splitting.

Figure 5 shows the overhead of NVTree with the modification for supporting conditional write. It has to scan all logs before inserting or updating. There is 19% slow down compare to its original version without conditional write. RNTree can support conditional write

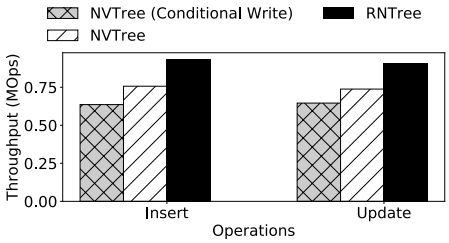


**Figure 5: Conditional overhead of NVTree**

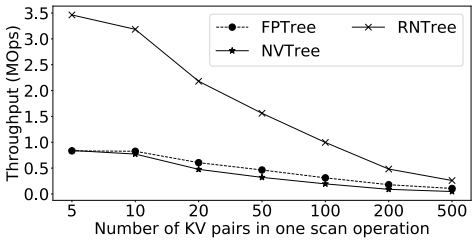


**Figure 6: Range query performance with different number of KVs handled in a range query operation.**

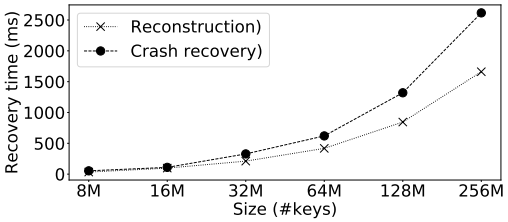


**Figure 7: Recovery time for RNTree with different tree size.**

with no overhead as it naturally finds the position of the key with the help from slot array.

*6.2.3 Update and remove.* `update` and `remove` have similar property as `insert`. The performance is mainly determined by the number of persistent instructions. For example, FPTree has higher `remove` throughput than other trees as it only has one persistent instruction in a `remove` operation, i.e., resetting the bitmap.

*6.2.4 Mixed benchmark.* As short summary, RNTree has the best (or one of the best) `find`, `insert`, `update` performance and fair `remove` performance. We also did the mixed operation benchmark with each operation comprises 25% of all operations. RNTree can run faster than other works by 25% ∼ 44%.

*6.2.5 Range query.* Different from the `find` operation, the performance of range query operation does not only depend on traversing the tree. Instead, it mainly depends on how fast it scans leaf nodes after reaching the first leaf node containing the start key. The performance of range query mainly depends on how many KVs it queries each time.

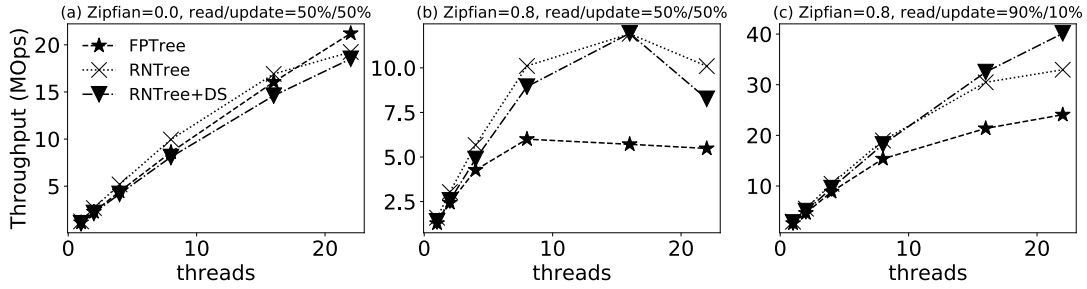


**Figure 8: Throughput of different trees with increasing number of threads.**

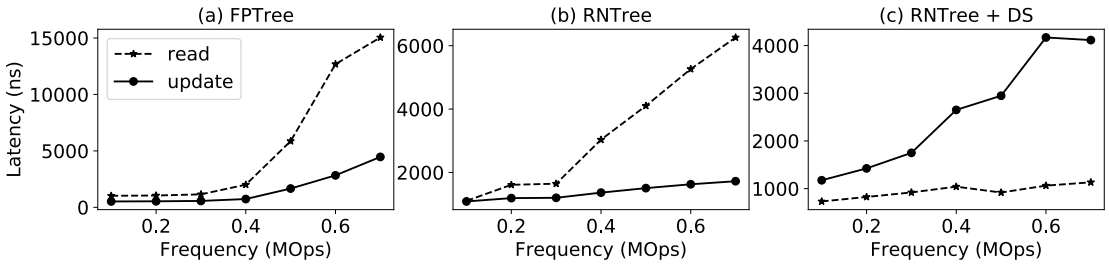


**Figure 9: Latency in different request frequency. There are 24 concurrent workers submitting requests with 50% read and 50% update. Keys' distribution obeys Zipfian's law. The Zipfian coefficient is 0.8.**

RNTree does the scan by applying the filter function on each entry in leaf node. NVTree and FPTree meet some difficulties because of their unsorted leaf nodes. Each leaf node needs to be sorted first. Our implementations use the sort algorithm in C++ Standard Library. Figure 6 shows the results of range query operations. Here, the throughput means how many range query operations get executed. RNTree is about 4.2× faster than NVTree and FPTree for different number of queries.

*6.2.6 Recovery.* When reboot, the internal nodes will be reconstructed from the persisted leaf nodes (*reconstruction*). If RNTree has to recover from crashes, non-critical data like *nlogs* are inconsistent. They can be reconstructed by other persistent data i.e. the slot array and logs (*crash recovery*). For example, to reset *nlogs*, the number of log entries allocated, it needs to scan the slot array to find the max index of log entries.

We evaluate the recovery performance of RNTree for different tree sizes. Figure 7 depicts the results. Recovery time or reconstruction time has a linear correlation to the tree size. The time of crash recovery is about 60% higher than that of reconstruction.

## 6.3 Concurrent benchmarks

The concurrent benchmarks focus on the scalability comparison between RNTree and FPTree (the only existing work for multi-threading). We have two versions of RNTree, with or without the dual slot array design. The dual slot array design can benefit read operations, but requires extra instructions to copy and protect the slot array for write operations. In the following, we denote RNTree as RNTree without the dual slot array design, and use RNTree+DS for RNTree with the dual slot array design. We use YCSB-A [22] as the default benchmark, which is composed of 50% `update` and 50% `find` operations. To eliminate the possible variations, we bind every thread to exactly one core and fix the CPU frequency.

*6.3.1 throughput.* Figure 8 shows the result of scalability. We test three cases: (a) YCSB-A benchmark with uniform workloads, (b) YCSB-A benchmark with skewed workloads. The skewed workload obeys Zipfian's law [23] with coefficient 0.8. We hash keys to distribute hottest keys to different leaf nodes. (c) Skewed read intensive benchmark, with 90% reads and 10% updates. In Figure 8(a), we observe that in uniform workloads, both FPTree and RNTree has linear scalability, which agrees with the result reported in FPTree paper. In a tree-like structure, as modification usually happens in low-level nodes, workloads with uniform distribution have rare contentions.

However, workloads in the real world are usually skewed. In Figure 8(b), FPTree can only scale to 4 threads. FPTree has two main problems: (1) For `update` operation, it has to lock the leaf node during the whole operation. In skewed workload, hot leaf nodes will always be locked, (2) For `find` operation, it will always abort the transaction and traverse from the root again if the leaf node is locked by another `update` operation. Thus, `find` operation has a much worse performance with more conflicts. The RNTree and RNTree+DS have about 1.8× higher throughput than FPTree with 24 threads.

Figure 8(c) depicts the performance of read intensive workload. FPTree still can not scale well, as its `find` can be easily broken by `update` operations. RNTree has the same problem. But RNTree performs better, because it spends less time in critical sections, and

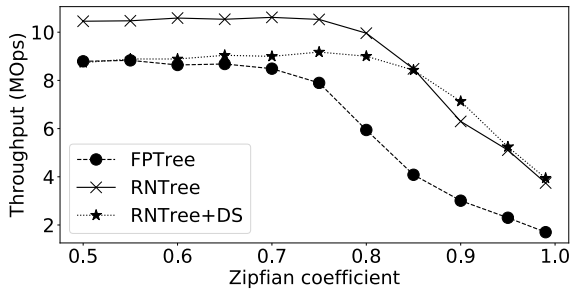


**Figure 10: YCSB-A performance (8 threads) with increasing contention controlled by the Zipfian coefficient.**

`update` operations have less possibility to break `find` operations. Different from others, RNTree+DS has near linear scalability.

*6.3.2 Latency.* The advantage of dual slot array design is not totally exposed in the throughput experiments. As the benchmark requires all workers to submit requests one by one, faster read incurs more frequent update, leading to more contentions, which conversely reduce the total throughput. In latency experiments, we limit the frequency of each workers submitting their requests and analyze the latency of read and update operations. With the increase of request frequency, the latency increases. The result is in Figure 9.

For FPTree, the latency of read can be as high as 15 $\mu$s, and the latency of update is about 5 $\mu$s when contention is higher. The read latency of RNTree is also high, about 6 $\mu$s. But its update latency can be limited within 2 $\mu$s. For RNTree+DS, because of its dual slot array, its read latency is below 1 $\mu$s. Thus, Figure 9 can indicate the true property of three trees. RNTree has better update performance. RNTree+DS has much better read performance, with a little sacrifice of update. FPTree performs poor for both operations.

*6.3.3 Skewness.* We also evaluate the performance for different skewed workloads. Figure 10 shows the throughput of RNTree and FPTree, using 8 threads with different Zipfian coefficients vary from 0.5 to 0.99. Results with Zipfian coefficients of $[0, 0.5)$ are excluded because they have negligible contention and have similar performance to Zipfian=0.5. We can find that when the Zipfian is larger than 0.7, the performance of FPTree drops quickly. RNTree is less sensitive to contention. RNTree can be up to 2.3× faster.

## 7 CONCLUSION

This paper presents RNTree, a scalable NVM-based B+tree. The design of RNTree follows two principles. (1) it solves the trade-off between persistent overhead and sorted leaf nodes by using hardware transactional memory to increase the atomic-write size. (2) By removing heavy persistent instructions from critical sections and proposing a dual-slot design, the critical section is more efficient. The scalability can then be improved. Our evaluation results show that RNTree has 1.44×/4.2× better performance for single-key-operation/`scan` in single-thread evaluation, and 2.3× better performance in multi-thread evaluation.

## 8 ACKNOWLEDGEMENT

We thank anonymous reviewers for their valuable feedback. This work is supported by National Key Research & Development Program of China (2016YFB1000504), Natural Science Foundation of China (61433008, 61373145, 61572280).

## REFERENCES

[1] 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.

[2] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTM: Building Durable Transactions with Decoupling for Persistent Memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 329–343.

[3] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.

[4] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 105–118, 2011.

[5] E. R. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–14.

[6] 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*. ACM, 2016, pp. 371–386.

[7] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[8] 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 *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 167–181.

[9] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, 2011, pp. 21–31.

[10] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory." in *FAST*, vol. 11, 2011, pp. 61–75.

[11] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 91–104, 2017.

[12] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 15.

[13] Z. Wang, H. Qian, J. Li, and H. Chen, "Using restricted transactional memory to build a scalable in-memory database," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 26.

[14] M. Wilcox, "Add support for nv-dimms to ext4," https://lwn.net/Articles/613384/.

[15] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 478–496.

[16] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTX: Durable Transactions Made Decoupled," *ACM Trans. Storage*, vol. 14, no. 1, pp. 7:1–7:28, Apr. 2018. [Online]. Available: http://doi.acm.org/10.1145/3177920

[17] "Redis," https://redis.io/, 2018.

[18] B. PostgreSQL, "Postgresql," *Web resource: http://www.PostgreSQL.org/about*, 1996.

[19] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[20] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *International Symposium on Distributed Computing*. Springer, 2016, pp. 313–327.

[21] 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*. ACM, 2012, pp. 183–196.

[22] 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*. ACM, 2010, pp. 143–154.

[23] G. K. Zipf, *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio Books, 2016.