
# TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering

R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, and Sumit Kumar Monga, *Virginia Tech;* Hee Won Lee, *Samsung Electronics;* Minsung Jang, *Peraton Labs;* Ajit Mathew and Changwoo Min, *Virginia Tech*

https://www.usenix.org/conference/atc21/presentation/krishnan

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

# TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering

R. Madhava Krishnan Wook-Hee Kim Xinwei Fu Sumit Kumar Monga
Hee Won Lee* Minsung Jang† Ajit Mathew‡ Changwoo Min
*Virginia Tech* *Samsung Electronics** *Peraton Labs†*

## Abstract

We propose TIPS– a framework to systematically make volatile indexes persistent. TIPS neither places restrictions on the concurrency model nor requires in-depth knowledge of the volatile index. TIPS relies on novel *DRAM-NVMM tiering* to support an index-agnostic conversion, durable linearizability and its concurrency model called the *tiered concurrency* to achieve a good performance, scalability. TIPS proposes a hybrid low overhead logging technique called the *UNO logging* to guarantee crash consistency and to handle persistent memory leaks across crashes. We converted seven volatile indexes with different concurrency models and the Redis key-value store application using TIPS and evaluated them using YCSB. Our evaluations show that TIPS-enabled indexes outperform the state-of-the-art index conversion techniques PRONTO, NVTraverse, RECIPE, and the NVMM-optimized B+Tree indexes (BzTree, FastFair), Hash indexes (CCEH and Level Hash) and Trie (WOART) indexes by 3-10× while supporting strong consistency and index-agnostic conversion.

## 1 Introduction

Indexes are a fundamental building block in many storage systems, and it is critical to achieving high performance and reliability [34, 47]. With the advent of Non-Volatile Main Memory (NVMM), such as Intel Optane DC Persistent Memory [10, 52], there have been a numerous number of research efforts targeted towards developing NVMM-optimized indexes [5,11,13,26,30,36,42–44,53–56,58,59,63,65,66]. Such index designs mainly focus on reducing the crash consistency overhead by primarily relying on index-specific optimizations to improve the overall performance.

However, maturing and hardening an index requires a lot of time and effort. For example, recently proposed NVMM indexes have critical limitations, such as (1) weaker consistency guarantee [23, 37], (2) not handling memory leaks in the wake of a crash [11, 26, 36, 53, 66], (3) limited concurrent access [13, 27, 36], and (4) not supporting variable-length keys [13,26,55,65]. Most of these missing features are critical in real-world systems and it delimits the adoption of these indexes to the real-world applications without further maturing.

Alternatively, there are decades of research on in-memory DRAM indexes [16, 39, 45, 47, 60] which are well optimized, engineered and used in many real-world applications such as in-memory key-value stores and databases [18, 19, 32, 57]. If we can leverage these in-memory DRAM indexes for NVMM, it not only gives a large pool of well-engineered indexes but it will also pave way for the real-world applications built on top of these indexes to use and adopt NVMM. The challenges in building an NVMM-optimized index has lead to a spike in the interest to port legacy DRAM applications, particularly in-memory key-value stores [1–4, 6, 7, 46, 49]. However, prior works [6, 46, 49] report that manual porting is complex, and error-prone requiring a lot of engineering effort. So we believe that *it is important to provide a systematic way to convert DRAM-based indexes for the NVMM*.

A few recent studies have proposed techniques [50, 62] or guidelines [15, 21, 23, 37] to convert volatile indexes to NVMM. Unfortunately, these techniques have some critical limitations such as (1) limited applicability due to restrictions on the concurrency control (*e.g.*, supporting only lock-free indexes) [15,21,23,37,50], (2) supporting a weaker consistency guarantee (*i.e.*, buffered durable linearizability) [23,37,62], (3) requiring in-depth index-specific knowledge [15, 21, 23, 37], (4) high performance and storage overhead due to their crash consistency mechanism [23, 50, 62], and (5) not addressing persistent memory leaks [15,21,23,37,62]. We further discuss the limitations of these techniques in §2.

To address these problems, we propose TIPS– an index-agnostic framework to systematically make a volatile DRAM index persistent, while supporting (1) wider applicability by not placing any restrictions on the concurrency model of an index, (2) strong consistency model (*i.e.*, durable linearizability), (3) index-agnostic conversion without requiring in-depth knowledge on a DRAM index, (4) low overhead crash consistency mechanism, (5) safe persistent memory management (*e.g.*, no persistent memory leak), and in addition to achieving (6) high performance and good multicore scalability. This paper makes the following contributions:

- We propose a novel *DRAM-NVMM data tiering approach* and its concurrency model called the *tiered concurrency* to achieve good performance, scalability, and applicability.
- We propose a low overhead hybrid logging technique called the *UNO logging* to guarantee crash consistency, to prevent memory leaks and to guarantee durable linearizability.
- We propose the TIPS *framework* based on these approaches. TIPS provides index-agnostic conversion and does not place any restrictions on the concurrency model or require in-depth knowledge of the volatile index being converted.

*†The authors contributed to this work while they were at AT&T Labs Research. ‡The author is currently in Amazon.

- We converted seven volatile indexes with different concurrency models, a real-world key-value store Redis and evaluated them using YCSB [14]. Our evaluation shows that TIPS outperforms the state-of-the-art index conversion techniques and the NVMM-optimized indexes by 3-10× across the different YCSB workloads.

## 2 Background and Motivation

We discuss the limitations of existing conversion techniques and their implications below:

**(1) Restrictions on Concurrency Control.** All prior techniques have limited applicability; *i.e.*, they are designed to support only a specific concurrency model. For example, NVTraverse [21] and link-and-persist [15] are designed for lock-free indexes while MOD [23] is designed for purely functional data structures. PRONTO [50] is applicable only to the globally blocking indexes (*e.g.*, protected by a global mutex) while RECIPE [37] guidelines apply only to the indexes that support lock-free or fine-grained locking.

**(2) Supporting Weak Consistency.** Another limitation is that most techniques [23, 37, 62] support only a weaker consistency; A linearizable DRAM index converted using these techniques will support only a weaker consistency model, Buffered Durable Linearizability (BDL) [29]. Linearizability has been the standard consistency model in DRAM indexes for almost three decades [25]. Its NVMM counterpart is Durable Linearizability (DL) [29], and it plays a critical role in ensuring the correctness and consistency of the NVMM index and data. Indexes with a BDL guarantee can experience a large amount of data loss in the wake of a crash. Moreover, it increases the programming complexity as the developers are burdened with reasoning about the data consistency. This makes the conversion process complex and more error-prone; for instance, many fundamental and non-trivial crash consistency bugs have been found in the RECIPE indexes [21, 22].

**(3) Requiring In-depth Knowledge.** Many techniques [15, 21, 23, 37] require in-depth knowledge of the volatile index and expertise in NVMM programming to apply their guidelines correctly. Such efforts are non-trivial as even missing a single `sfence` or a `clwb` may render an index irrecoverable.

**(4) High Storage and Performance Overhead.** Many techniques suffer from high storage and performance overhead [23, 50, 62] mainly due to their crash consistency mechanism. For example, PMThreads [62] requires two full replicas (one each on DRAM and NVMM) of the original data. MOD [23] uses Copy-on-Write (CoW) to guarantee crash consistency. Such a high storage overhead might be acceptable for primitive data structures (*e.g.*, stack, vector). However, it can be detrimental for indexes and real-world key-value stores designed to handle a large volume of data. Although PRONTO [50] uses operational logging for crash consistency and employs a dedicated background thread for every writer to perform the logging, it still incurs a high overhead due to the synchronous waiting between the writers and background threads. We further empirically analyze these overheads in §7.

**(5) Persistent Memory Leaks.** Another critical but a largely understated problem is the persistent memory leaks. While the prior conversion techniques and the NVMM-optimized indexes focus on providing crash consistency, they completely ignore the memory leak problem. Although NVMM allocators can guarantee failure atomicity for allocation/free even the mature allocator such as the PMDK [28] does not provide an efficient solution to identify and fix the memory leak [17]. Hence it is critical to address this within the TIPS framework.

## 3 Overview of TIPS

We first discuss our design goals followed by the design overview. We assume that an index being converted has key-value store style operations such as insert, delete, update, lookup, and scan. Throughout this paper, we address insert, delete and update as writes, and lookup and scan as reads. The term *plugged-in index* denotes a user-defined volatile index on NVMM that is plugged into TIPS framework. We assume that locks can be reinitialized after crash recovery.

### 3.1 Design Goals

$G_1$ **Support Various Concurrency Models.** We aim not to place restrictions on the concurrency model of the volatile indexes. With TIPS, we support the conversion of indexes that use a global lock (*e.g.*, Mutex), lock-free (*e.g.*, CAS), and fine-grained locking (*e.g.*, ROWEX [39]).

$G_2$ **Support Durable Linearizability (DL).** The challenge to guarantee DL in TIPS is to achieve it without compromising the performance, scalability, and index-agnostic conversion.

$G_3$ **Support Index-agnostic Conversion.** We aim to support near black-box, index-agnostic conversion to circumvent the need for in-depth knowledge on the volatile index and NVMM programming. This will make the conversion process simple, intuitive, and less error-prone. We aim to achieve this by internally handling the complications of NVMM programming such as guaranteeing crash consistency and preventing persistent memory leak within TIPS and also providing an uniform programming interface to assist the conversion.

$G_4$ **Design a Low Overhead Crash Consistency mechanism.** TIPS can not rely on index-specific optimizations for crash consistency to support an index-agnostic conversion. The crash consistency mechanism should incur low overhead to achieve high performance and scalability. A crash consistency mechanism also should prevent persistent memory leaks by reclaiming the unreachable objects upon recovery.

$G_5$ **Achieve High Performance and Scalability.** Ideally, we aim to perform and scale on par with NVMM-optimized indexes. Also, we strive to retain the original characteristics of the plugged-in index; for example, if the volatile index is designed for cacheline efficiency or optimized for scalability, we aim to retain and leverage such characteristics to improve

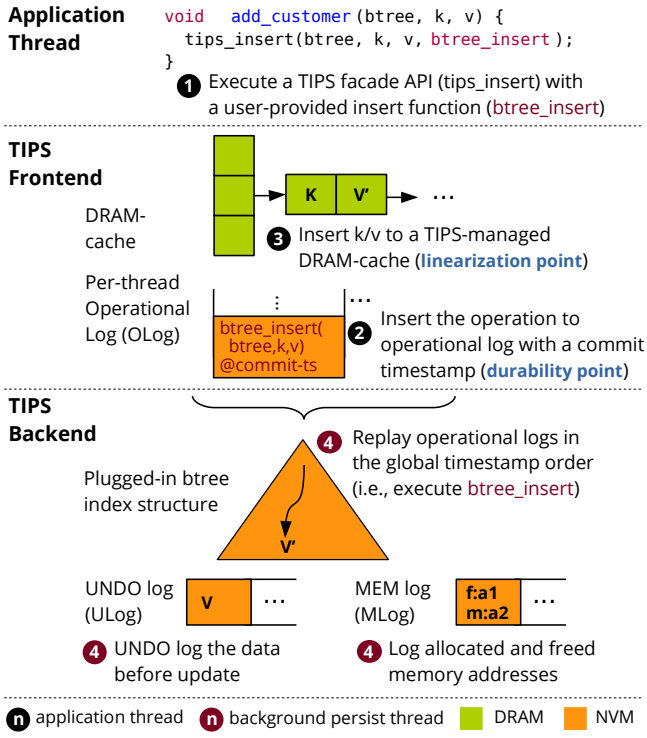


**Figure 1:** Illustrative example of inserting a key-value pair in TIPS.

the overall performance and scalability of a TIPS index.

## 3.2 Design Overview

### 3.2.1 High-Level Idea of TIPS

Figure 1 presents the TIPS architecture and illustrates how a write operation is handled in the TIPS framework. TIPS frontend consists of a hash table on DRAM (DRAM-cache) and an operational log (OLog) on NVMM. TIPS backend consists of the plugged-in index – the user-provided volatile index – a background thread (persist-thread), UNDO log (ULog), and MEM log (MLog). When a write is issued (❶), TIPS first commits it to the per-thread OLog for guaranteeing durability (❷ - durability point) and then inserts a new key-value pair entry (or a tombstone for deletion) in the DRAM-cache to make the write visible (❸ - linearization point). Then the persist-thread in the TIPS backend (❹) replays the same write in the background to update the plugged-in index. To guarantee crash consistent update to the plugged-in index, TIPS uses ULog to store the unmodified data for recovery. Once the plugged-in index is updated, the corresponding key-value entry in the DRAM-cache will be reclaimed later.

Alternatively, a lookup operation first looks for the target key in DRAM-cache and it goes to the plugged-in index only if the target key is not present in DRAM-cache. With this high-level idea, we next present the design overview of TIPS and also explain how it contributes towards achieving our design goals discussed in §3.1.

### 3.2.2 DRAM-NVMM Tiering ($G_1$, $G_2$, $G_3$, $G_5$)

At its core, TIPS adopts a novel DRAM-NVMM data tiering approach. As illustrated in Figure 1, two critical components that enable the DRAM-NVMM tiering are DRAM-cache in the TIPS frontend and plugged-in index in the TIPS backend. Although prior NVMM-optimized B+tree index designs [42, 55] and conversion techniques [50, 62] have proposed to use both NVMM and DRAM, our approach is fundamentally different. For example, PRONTO [50] builds the index on DRAM and logs the index operations on the NVMM to guarantee durability. Such a tiering design limits PRONTO indexes from scaling beyond the DRAM capacity. Instead, in TIPS, we propose tiering of the data (*i.e.*, key-value pairs) while keeping the plugged-in index intact on the NVMM.

**Benefits.** (1) Tiering the data between DRAM-NVMM makes our approach *generically applicable* to any index. This enables all the TIPS indexes to take advantage of the faster DRAM. (2) Unlike PRONTO, TIPS can achieve a better capacity scaling by keeping the plugged-in index on the NVMM. (3) The writes are made visible through DRAM-cache (❸ in Figure 1); this enables TIPS to guarantee DL agnostic of the plugged-in index. (4) On top of DRAM-NVMM tiering, we build the *plug-in programming model* to enable index-agnostic conversion (§3.2.6). (5) Tiering the data enables a new concurrency model called the *tiered concurrency*, which is key to achieving high performance and scalability.

### 3.2.3 Tiered Concurrency for Scaling Frontend ($G_1$, $G_5$)

Having TIPS frontend and backend enables two different levels of concurrency: (1) concurrency model of the DRAM-cache and (2) concurrency model of the plugged-in index. We call this a *tiered concurrency model*. TIPS frontend allows parallel readers and parallel disjoint writers as DRAM-cache is a concurrent hash table and OLog is per-thread. In the critical path, all requests succeeding in the TIPS frontend (all writes and read hits) will follow the concurrency model of DRAM-cache, and the operations that go to the TIPS backend (read misses and scan) will follow the concurrency model of a plugged-in index. Also, the background writes (❹ in Figure 1) to the plugged-in index is done off the critical path adhering to concurrency model of the plugged-in index.

In a nutshell, to achieve write scalability TIPS restricts writes to the faster frontend (DRAM-cache and per-thread OLog) and for read scalability, it relies on both the DRAM-cache and the concurrency model of the plugged-in index. Range scans always go to the plugged-in index as it requires full key-value data (see details in §4.1.6). Relying on the plugged-in index for read/scan helps TIPS to reduce the DRAM footprint as it does not need to cache the entire dataset in DRAM-cache. Instead, it can reclaim the keys once the plugged-in index is updated.

**Benefits.** (1) Even if the plugged-in index supports only blocking concurrency (*e.g.*, mutex), it can still leverage the

```c
/* TIPS facade API to use a TIPS-enabled index */
bool tips_insert(void *ds, key_t k, value_t v, fn_insert_t *f);
bool tips_update(void *ds, key_t k, value_t v, fn_update_t *f);
bool tips_delete(void *ds, key_t k, fn_delete_t *f);
value_t *tips_lookup(void *ds, key_t k, fn_lookup_t *f);
value_t *tips_scan(void *ds, key_t start_key, int range,
                   fn_scan_t *f);

/* TIPS plug-in API to implement a TIPS-enabled index */
bool  tips_ulog_add(void *addr, size_t size);
void* tips_alloc(size_t size);
void  tips_free(void *addr);
```

**Figure 2:** TIPS *facade APIs* to access a TIPS-enabled index, and *plug-in APIs* for plugging-in a volatile index to TIPS.

DRAM-cache to process all the writes and read hits concurrently to achieve good performance. (2) Unlike the previous techniques [15, 21, 37, 50], it does not place any restrictions on the concurrency model of the volatile indexes and hence any volatile index can be plugged-in to the TIPS framework.

#### 3.2.4 Adaptive Scaling for Backend Scalability ($G_5$)

The backend writes are slower than the frontend as it writes to the NVMM. This can cause an imbalance in the system; to prevent this, we propose adaptive scaling of background writers (workers) for scaling the TIPS backend. With adaptive scaling, TIPS continuously monitors both the frontend and backend write throughput, and when there is an imbalance it scales up the worker count to catch up with the faster frontend and vice versa. While scaling up, TIPS identifies the best worker count based on the write scalability of the plugged-in index and it caps the scale-up at that count to get maximum performance. Also, the real-world workloads are rarely 100% writes, so the time between writes and the adaptive scaling can help the backend writers to catch up with a faster frontend.

**Benefits.** (1) It effectively utilizes the write concurrency of the plugged-in index. (2) Unlike PRONTO [50] which demands a dedicated worker for every foreground writer, TIPS can dynamically adjust the worker count based on nature of the workload and the plugged-in index.

#### 3.2.5 UNO Logging for Crash Consistency ($G_4$)

To achieve a low overhead index-agnostic crash consistency, we propose the *UNO logging protocol*, which makes a hybrid and synergistic use of traditional **UN**DO logging and **O**perational logging. In TIPS, we use operational logging (OLog) to guarantee immediate durability and UNDO logging (ULog) to ensure failure-atomicity while updating the plugged-in index. The unique aspect of *UNO* logging lies in how we leverage the OLog to reduce the notorious UNDO logging overhead and also reduce the number of `p-barrier` (`clwbs` followed by `sfence`) by batching the recurring updates to the same cache line and consequently achieve a low overhead crash consistency. Furthermore, MEM Log (MLog) internally logs all the allocated and freed addresses and TIPS uses this information to identify and free all the unreachable memory upon recovery to prevent memory leaks and defers the actual memory free operations until OLog entries are consumed to prevent double-free bugs.

```c
1  void hash_insert(hash_t *hash, key_t key, val_t value) {
2      node_t **pprev_next, *node, *new_node;
3      int bucket_idx;
4      pthread_rwlock_wrlock(&hash->lock);
5      // Find a node in a collision list
6      // Case 1: update an existing key
7      if (node->key == key) {
8          // Before modifying the value, backup the old value
9  +       tips_ulog_add(&node->value, sizeof(node->value));
10         node->value = value; // then update the value
11         goto unlock_out;
12     }
13     // Case 2: add a new key
14     // Allocate a new node using tips_alloc
15 +   new_node = tips_alloc(sizeof(*new_node));
16     new_node->key   = key; new_node->value = value;
17     new_node->next  = node;
18     // Backup the prev node before modifying it
19 +   tips_ulog_add(pprev_next, sizeof(*pprev_next));
20     *pprev_next = new_node; // then update then the node
21 unlock_out:
22     pthread_rwlock_unlock(&hash->lock);
23 }
```

**Figure 3:** Code snippet of a TIPS-enabled hash table insert. Only three lines are modified in the original code; Lines 9, 19 for UNDO logging and Line 15 for persistent memory allocation.

**Benefits.** (1) Using OLog requires only *two* `p-barriers` in the critical path for all write operations; *one* `p-barrier` to persist a OLog record and *one* more to persist the tail pointer (§4.1.5). This makes the durability guarantee cheap and consequently a better performance (§7.2, §7.4). (2) TIPS alleviates the UNDO logging overhead by leveraging the OLog information to selectively log only the memory required for correct recovery besides the merit that UNDO logging in TIPS is performed by the background workers (§4.3.2). (3) With MLog TIPS handles the persistent memory leaks with its framework instead of delegating it to the users.

#### 3.2.6 Plug-In Programming Model for Index-agnostic Conversion ($G_2$)

The plug-in programming model provides two sets of APIs as shown in Figure 2: (1) plug-in APIs to plug-in a volatile index to the TIPS framework and (2) facade APIs to access the plugged-in index. The facade APIs internally manage the OLog and DRAM-cache without requiring any user intervention (steps ❶, ❷, ❸ in Figure 1). With the facade APIs, the plugged-in index implementation should be passed as a function pointer `f` which is used by TIPS to update the plugged-in index. To guarantee crash consistent updates to the plugged-in index, the developers must modify their index implementation using TIPS plug-in APIs. The modifications are simple, as shown in Figure 3: (1) replacing the volatile memory allocation (and free) with `tips_alloc` (and `tips_free`) and (2) adding `tips_ulog_add` before modifying/writing to an NVMM address. TIPS will execute this modified code (*e.g.*, *hash_insert* in Figure 3) during the background reply.

By replacing the *malloc* with `tips_alloc`, the plugged-in index will now be allocated on the NVMM, and `tips_alloc` will also capture all the newly allocated address in the MLog to prevent persistent memory leaks. The developer

added `tips_ulog_add` would ensure crash consistency while updating the plugged-in index, and TIPS internally optimizes the UNDO logging (§4.3.2) for better write coalescing and performance. Moreover, a developer is not required insert `p-barriers` to the volatile codebase manually. Instead, they just need to annotate the stores to NVMM with `tips_ulog_add`. Thus, LoC changes in the plugged-in indexes are minimal, as shown in Table 2. While the developers still have to add the `tips_ulog_add` maually, they need not handle the persistence/visibility order as in the case of manually inserting `p-barriers`, and this makes the conversion easy and less error-prone. Note that updates to the newly allocated addresses (in Memlog) and existing addresses (in ULog) are batched and persisted internally by TIPS before reclaiming the respective logs. More details on this in §4.3.

# 4 Design of TIPS

We first describe the TIPS frontend design in §4.1, followed by the TIPS backend design in §4.2.

## 4.1 TIPS Frontend Design

### 4.1.1 DRAM-cache

The DRAM-cache is a concurrent open chaining hash table. Concurrent writers working on the different buckets are allowed to proceed in parallel while the ones working on the same bucket are synchronized using a spinlock. Readers do not need any synchronization (*i.e.*, lock-free reads) as all writes to DRAM-cache are performed via a single atomic store. DRAM-cache also employs a RCU-style epoch based reclamation scheme for garbage collection. We choose open chaining hash table to guarantee $O(1)$ writes and avoid expensive rehashing in the critical path. We discuss the configurations for chain length and bucket size in §6 and §7.

### 4.1.2 Handling Write Operations

All write operations are first committed to the OLog to guarantee durability, and then a new entry is created and added to the DRAM-cache to make the writes visible. For delete, the entry also carries a tombstone mark for the readers to identify that it has been deleted logically. To make writes faster, the entries are always added at the head of a bucket. We explain how TIPS guarantees Durable Linearizability (DL) in §5.

### 4.1.3 Handling Lookup Operation

Readers first traverse the DRAM-cache bucket looking for the target key. As the collision chain is sorted by the arrival order of write requests, the readers can stop at the first match instead of traversing the entire chain. If the key is not present, then TIPS internally redirects the readers to the plugged-in index using the function pointer provided in the `tips_lookup` call. Scan operations require traversing the plugged-in index and the OLog to return a consistent result. We further describe it in §4.1.6 after introducing the OLog design.

### 4.1.4 Safe Reclamation

Since all writes happen in the DRAM-cache, the collision chain can proliferate and result in a high chain traversal overhead. To address this, we employ a background garbage collector thread called the `gc-thread`. When the chain length of a bucket exceeds the preset threshold, the `gc-thread` is triggered, which then visits the respective bucket and safely reclaims the entries. To ensure safe reclamation of entries *i.e.*, without impacting the concurrent readers, TIPS employs an epoch-based reclamation scheme, which is widely used in lock-free and RCU-based data structures [20,24,48,61]. TIPS manages two types of epochs: (1) local epoch, which is the global epoch value when readers enter the critical section, and (2) global epoch, which is advanced when all active readers in the current epoch exit the critical section.

The `gc-thread` first logically reclaims entries by unlinking them from the collision chain and storing them in the free-list. The reclaimed entry then becomes invisible to new readers. Note that the `gc-thread` logically reclaims the entries that are successfully replayed, so upon a read miss, readers can still retrieve the reclaimed entries from the plugged-in index. To determine if there are any outstanding readers on the logically reclaimed entries, the `gc-thread` checks the local epoch of all the active readers. If the local epoch is the same as the global epoch, *i.e.*, no outstanding readers from the previous epoch exist; then the `gc-thread` physically frees the entries in the free-list that are logically reclaimed two epochs ago. Note that the `gc-thread` atomically modifies the collision chain, so the readers and writers are free to enter the critical section without waiting for reclamation to finish.

### 4.1.5 Operational Log (OLog)

OLog guarantees durability to the write operations executed on the DRAM-cache. An OLog record consists of the operation type (insert, delete or update), the respective function pointer to the plugged-in index logic, key, value, and a global timestamp (`commit-ts`) to denote the commit order of an operation. OLog is a circular buffer where new entries are always added at the tail. The atomicity for an OLog write is guaranteed by its tail pointer update. Once an OLog record is written and persisted, the tail pointer is atomically updated to point to the new record, followed by persisting the tail pointer.

### 4.1.6 Handling the Scan Operations

**Algorithm.** The scan operation in TIPS is always directed to the plugged-in index, as it requires a full range of keys and values. However, the plugged-in index is not guaranteed to be up-to-date since the persist-thread might still be propagating some of the updates that might potentially fall within the scan range. So, after scanning the plugged-in index, TIPS traverses the OLogs looking for any potential keys that might fall within the scan range. Upon finding any, the scan buffer is adjusted to incorporate not yet propagated operations.

**Traversing OLog.** To minimize the OLog traversal over-

head, we leverage the commit timestamp of each OLog record (`commit-ts`) and the timestamp when a scan operation starts (`scan-ts`). While traversing the OLog, the scan thread compares its `scan-ts` with the `commit-ts`. If the `commit-ts` $\leq$ `scan-ts`, then the scan thread reads the key information in the OLog record and checks if it falls within the scan range. It is safe to ignore OLog records with `commit-ts` $>$ `scan-ts` because these are in the future with respect to the scan thread, so it can stop traversing the OLog. Refer to §5 for correctness. Traversing the OLog is fast and adds only a negligible overhead for three reasons: (1) We partially traverse not-yet-propagated OLog entries stopping at the first future entry. (2) As the persist-thread continuously propagates the updates, there will not be much backlog in the OLog. (3) Finally, the scanning of OLog is a sequential read operation on the NVMM, which is almost as fast as reading from the DRAM [64]. We verify this emperically in §7.3.

### 4.2 TIPS Backend Design

The primary role of the TIPS backend is to combine and replay the per-thread OLog on the plugged-in index. To guarantee correct replay order, the persist-thread combines the per-thread OLog entries, and then it spawns the required number of background workers, which replays the combined entries to the plugged-in index. The persist-thread decides the required worker count on-fly using the adaptive scaling algorithm. The following subsections explain each of these steps in detail.

#### 4.2.1 Adaptive Scaling of Background Workers

TIPS automatically adjusts the number of workers at every epoch based on the write scalability of the plugged-in index and the nature of the workload. One epoch ($e$) is defined as one iteration of combining and replying the OLog entries. We denote $W_{e+1}$ as the worker count for the next epoch $e+1$.

At every epoch $e$, TIPS calculates the foreground throughput ($F_e$), which is the number of OLog entries produced by application threads during the epoch, and the background throughput, which is the number of OLog entries consumed by the worker threads during the epoch. TIPS calculates the processing rate $R_e$, which is $F_e/B_e$. It aims to maintain $R_e$ close to 1 by adjusting the number of workers ($W_{e+1}$).

If $R_e > 1$, the foreground writers are filling up the OLog at a faster pace, and if this situation persists, it will lead to blocking of writers as the workers are slow in clearing up the OLogs. So TIPS will increase $W_{e+1}$ by a predefined step $\Delta$ aiming to improve the $B_e$ and keep up with $F_e$.

If $R_e < 0.5$, workers are clearing up the OLogs faster than foreground writers fill them up. Employing excess number of workers is a waste of CPU, so TIPS decreases $W_{e+1}$ by $\Delta$.

Finally, if $1 < R_e < 0.5$, TIPS considers that the workers are on par with foreground writers and hence maintains the same number of workers (*i.e.*, $W_{e+1} = W_e$).

TIPS maintains a user-configurable upper bound ($W^{up}$) and a lower bound ($W^{low}$) to cap the number of workers ($W_e$) while scaling up and down respectively. In addition, while scaling up, TIPS memorizes the best performing worker count ($W^{max}$); so that if $W_e$ reaches the upper bound TIPS can fall back to $W^{max}$ and continue until scaling down is needed.

By default, TIPS sets $W^{low}$ to 0 and $W^{up}$ to the number of physical cores. TIPS uses a smaller $\Delta$ when the worker count is small (*i.e.*, $\Delta = 1$ when $W_e < 4$) and uses a bigger $\Delta$ when the worker count is large (*i.e.*, $\Delta = 4$ when $W_e \geq 4$).

#### 4.2.2 Concurrent Replay of OLog Entries

After deciding the number of workers, persist-thread combines the per-thread OLog entries and adds them to the per-worker queue. To avoid copy overhead, TIPS maintains only a pointer to the OLog record in per-worker queue.

There are two key invariants that must be maintained in the combining process: (1) Since the OLog records are replayed concurrently, it is essential to maintain the correct ordering, especially for non-commutative operations (*e.g.*, `insert(k1,v)` and `delete(k1)`). For commutative operations (*e.g.*, `insert(k2,v)` and `delete(k3)`), the replay can be done in any order without violating the correctness. (2) The OLog can not be reclaimed until all the entries in the per-worker queues are consumed. For (1), TIPS uses hashing to ensure that all the non-commutative operations will be placed in the same worker-queue. After combining, persist-thread spawns $W_e$ workers. Then each worker sorts the entries in the timestamp (`commit-ts`) order and replays them to the plugged-in index. This ensures that non-commutative operations are always executed by the same worker in their exact order of arrival. For (2), persist-thread waits until the end of the current epoch to safely reclaim the OLog (see the details in §4.3.3).

### 4.3 UNO Logging

In this section, we describe the design of ULog and MLog. Similar to OLog, the atomicity of ULog and MLog writes is guaranteed by atomic tail pointer update. Both ULog and MLog are protected using a global Readers-Writer lock.

#### 4.3.1 Memory Log (MLog)

**What to log?** All the newly allocated and freed addresses are recorded in the MLog along with a tag to denote if the address is allocated or freed. Also, each allocated address carries a timestamp (`alloc-ts`) to denote the time at which the particular address is allocated. TIPS memory allocation APIs internally use PMDK allocator. PMDK not only guarantees failure atomicity for memory allocation and free but also atomic persistence of a variable that stores the NVMM heap address [28]; TIPS passes an address in MLog to the PMDK memory allocation API that guarantees the address pointing to the allocated memory is persisted when returning from the API. Similarly, PMDK memory free guarantees atomic persistence for an address that is set to `NULL`. During recovery, all the non-`NULL` addresses with the "allocated" tag in MLog are deemed to be non-reachable. Such addresses are freed to avoid memory leaks as the insert operations that created these

addresses will be re-executed again from the OLog. Similarly, it is possible to re-execute the same OLog entry more than once; This can cause a double-free bug if a crash happens amidst a delete operation. To avoid this, TIPS logically removes the address from the plugged-in index, stores it in the MLog, and defers the actual memory free until the subsequent OLog reclamation. Because after reclaiming the OLog, the delete operation can not be re-executed again.

**A running example.** Suppose that inserting (or deleting) a key triggers split (or merge) on the leaf node `A` in a B+-tree, and a new leaf node `A'` is allocated (or the existing `A` freed). Say a crash happens before the completion of split (or merge) but after allocating `A'` (or freeing `A`). During the recovery, TIPS will re-execute the same insert (or delete) from the OLog, and it once again allocates a new leaf node `A"` (or free `A` again). This scenario leads to a persistent memory-leak of `A'` (or double-free of `A`). With the MLog, TIPS can reclaim the previously allocated node `A'` (or restore node `A`) during recovery to avoid persistent memory leak (or double-free).

#### 4.3.2 UNDO Log (ULog)

**What to log?** ULog is used by the worker threads to guarantee failure-atomic updates to the plugged-in index. Generally, all the addresses that are being modified are required to be logged in the ULog. Instead, in TIPS, we leverage the OLog information to selectively log only the addresses that are needed for correct recovery. To decide whether to log a given address, the workers rely on two timestamp information: 1) the time at which the requested address is allocated (`alloc-ts`) and 2) the time of last OLog reclamation (`reclaim-ts`). If the requested address has its `alloc-ts` > `reclaim-ts` (*i.e.*, the address is allocated after the last OLog reclamation), then the operation to recreate the contents of this address is guaranteed to be present in the OLog. Hence the workers skip the UNDO logging. Otherwise, the workers first check if the requested address is already logged due to any previous write request. If so, the workers will skip the UNDO logging; else, they record the contents of the requested address in the ULog. The persist-thread defers the persistence of addresses in the ULog until the subsequent ULog reclamation. Thus, recurring updates to the same address can be batched and persisted at the start of every ULog reclamation.

**A running example.** Say a B+-tree node `A` is being modified 500 times between two ULog reclamations. Then a worker will log `A` in the ULog at the time of its first modification and reuse the same record until the next ULog reclamation. After the 500th update, say a ULog reclamation is triggered; the persist-thread before reclaiming the ULog will persist `A` with its latest update. If a crash happens before persisting `A`, during the recovery, TIPS correctly spots and reverts node `A` to the state before its first modification from the ULog. Then it re-executes all the 500 updates from the OLog to bring `A` to its latest state before the failure.

#### 4.3.3 UNO Logging Reclamation

Log reclamation is triggered when any of OLog or ULog reaches their preset capacity threshold. The persist-thread always reclaims all the logs together even if only one of them reaches its capacity threshold. That is because, in TIPS, the information required for a correct recovery is distributed across all three logs. The reclamation yields for two cases; it waits until the (1) current epoch of background replay to end, and (2) pending scans to finish traversing the OLog. The UNO logging reclamation consists of the following two steps:

**Step 1. Flush the addresses in the ULog and MLog.** First, all the addresses in the ULog and MLog are persisted by calling `p-barrier`. This guarantees that all the writes that occurred since the last UNO reclamation are persisted.

**Step 2. Reclaim the logs.** Replayed OLog entries and persisted ULog entries in Step 1 are obsolete; they can be safely reclaimed to free up space for the incoming writes. Since the OLog has been reclaimed; we no longer required to keep track of the allocated and freed addresses; so the logically reclaimed addresses stored in the MLog are physically freed and then the MLog space can also be safely reclaimed.

**Crash safety of reclamation.** The reclamation procedure is crash-safe. The crash safety is guaranteed by *atomically setting the `flush_done` flag after the completion of Step 1*. Upon a crash, the recovery procedure first checks the `flush_done` flag. If the flag is set, it means that Step 1 has been completed successfully before the crash occurred and this guarantees that all the updates to the plugged-in index are persisted. Hence the recovery simply reclaims the remaining logs (Step 2) and terminates. If the flag is unset, then a standard recovery procedure described in the following section is followed.

### 4.4 Recovery

TIPS flushes all the logs and sets the tail pointer of the UNO log to `NULL` upon a safe termination. Therefore, if the tail is non-`NULL` upon a TIPS restart, it triggers the recovery procedure. If `flush_done` is set, recovery proceeds as described in the previous section. Otherwise, the recovery consists of three steps; (1) replay ULog to set the index to the exact consistent state that existed at the last UNO reclamation; (2) free all the newly allocated addresses (before the crash) from the MLog to prevent persistent memory leaks; (3) replay the OLog to get to the last successfully committed update before the crash. Note that the logs are replayed only up to their tail pointers to avoid executing any partial writes during the recovery.

If a crash occurs during the ULog replay (Step 1) or MLog free (Step 2), TIPS can continue from where it left off. This is because the changes to the plugged-in index due to ULog replay are immediately persisted. Also, freeing MLog after ULog replay does not affect the persistent state of the index. As described in §4.3.1, freeing MLog entries is guaranteed with atomic persistence to `NULL`, making this step failure-safe. On the other hand, if there is a crash while executing OLog

entries (Step 3), TIPS treats it similar to the failure during normal execution; *i.e.*, after rebooting, TIPS re-executes all the three steps. Because replaying OLog during recovery is treated similar to replaying the OLog in the background during normal execution. Note that re-executing OLogs after the ULog and MLog replay is idempotent, so it does not affect the consistency of the plugged-in index.

## 5 Correctness of TIPS

**Theorem 1.** TIPS *guarantees Durable Linearizability (DL).*

*Proof.* To guarantee DL, TIPS must satisfy three main invariants: (1) Effect of a committed operation can not be undone in the face of a crash. For all the non-commutative operations (*e.g.*, `insert(k1,v)`, `delete(k1)`), (2) the order of commit (*i.e.*, OLog write), and visibility (*i.e.*, DRAM-cache write) must always be maintained; and (3) also the background replay must be performed in the linearization order–in the same order as the operations are made visible in the TIPS frontend.

For (1), the effect of a write operation is visible only after updating the DRAM-cache entry, which is strictly done after persisting the OLog record. This guarantees that the readers will never observe the effects of non-durable write. For (2), the commit and the visibility order for non-commutative operations are synchronized using the per-bucket spinlock in the DRAM-cache as such operations is always guaranteed to happen on the same DRAM-cache bucket. A writer acquires the lock and commits its operation in the OLog with a timestamp (`commit-ts`). Then it adds the entry in the DRAM-cache and releases the lock, which guarantees that the order of commit and visibility is always the same for non-commutative operations. For (3), as described in §4.2.2, TIPS uses hashing to ensure non-commutative operations always go to the same worker queue. Then the worker sorts its queue in the `commit-ts` order and updates the plugged-in index to maintain the linearization order. For commuting operations, maintaining the linearization order is not required as they work on disjoint keys, so the effect of such operations will be the same regardless of the order in which they are executed.

By guaranteeing DL, TIPS eliminates the possibility of non-trivial crash consistency bugs as found in the previous work RECIPE [37]. Particularly, TIPS avoids the dirty read bugs as unpersisted writes are never visible to readers as guaranteed by (1). Even if certain reads are served from the DRAM-cache and say a crash happens, the OLog reply during the recovery will ensure that the plugged-in index is up to date with all the committed writes that happened before the crash. This ensures that all the pre-crash reads are still valid as they can be retrieved from the plugged-in index. □

**Theorem 2.** TIPS *guarantees DL for scan operations.*

*Proof.* A scan operation in TIPS traverses both the plugged-in index and the OLog, and then it merges the results. The DL guarantee can be violated if (1) the scan thread reads a partially written OLog entry and (2) if it reads the unpersisted tail pointer. Both these cases can result in loss of data if a crash happens because of reading non-durable data. To avoid (1), the scan thread traverses the OLog from head to tail and this will hide any ongoing OLog writes as the failure atomicity of an OLog write is guaranteed by atomically updating the tail pointer (§4.1.5). To avoid (2), the scan thread before starting its traversal, checks if the tail pointer is persisted. If yes, it starts traversing; else, it backs off and retries. □

## 6 TIPS Implementation

TIPS is written in C, and the core library is about 5000 LoC. We use the hardware clock (`rdtscp` in x86 architecture) for scalable timestamp allocation. To address the clock skew between the CPU cores, we use ORDO [31] as done in many other previous works [33, 35]. Note that ORDO does not require any hardware extensions. We set the maximum DRAM-cache chain length to 5 beyond which the `gc-thread` starts reclaiming the applied entries in the chain. We chose this number after carefully considering the DRAM/NVMM random read performance ratio; random read latency in NVMM is about 5× slower than DRAM [64] so that traversing more than 5 nodes in the collision chain do not pay off.

## 7 Evaluation

We evaluate TIPS by answering the following questions: (1) How do TIPS perform against prior conversion techniques (§7.2)? (2) How do the TIPS-indexes perform against prior NVMM-optimized indexes (§7.3)? (3) How do the TIPS-indexes scale for different workloads (§7.4)? (4) What is the sensitivity for DRAM-cache and UNO logging size (§7.5)? (5) How does TIPS impact real-world application (§7.6)?

**Evaluation Platform.** We use a system with Intel Optane DC Persistent Memory (DCPMM). It has two sockets with Intel Xeon Gold 5218 CPU with 16 cores per socket, 512GB of NVMM (4×128GB) and 64 GB of DRAM (4×16GB). We used GCC 8.3.1 with `-O3` flag to compile benchmarks and ran all our experiments on Linux kernel 4.18.16.

**Configuration.** We used YCSB [14]– a standard key-value store benchmark (Table 1) for all our evaluations. We used index-microbench [60] to generate the YCSB workloads. We ran the benchmarks for 32 million keys; we first populate an index with 32M keys and then run the workloads, which performs 32M operations. We use random integer and string keys with uniform distribution. We also evaluate the TIPS for large datasets and Zipfian distribution in §7.4. We preset the size of our per-thread OLog and the global ULog to 32MB each, the DRAM-cache to cache 25% of the total number of keys (300 MB) and the upper bound for the number of workers ($W^{up}$) to 32 (*i.e.*, half of the available CPUs). We present the sensitivity analysis for these configurations in §7.5. To ensure a fair comparison, we carefully chose the indexes as the existing conversion techniques are specific to certain concurrency

models. We also ported all the indexes to use the PMDK memory allocator. Porting all the RECIPE [37] and NVTraverse [21] indexes with PMDK allocator incurred about 1800 LoC. For all our evaluations (unless mentioned specifically), we use 32 threads to match the maximum physical CPU cores (without hyperthreads) available on our platform and present the scalability results for all the TIPS-indexes in §7.4.

### 7.1 Converting Volatile Index using TIPS

Converting an index using TIPS is simple (§3.2.6), and it requires only minimal LoC changes as shown in Table 2. For all the indexes, we replace the memory allocation (and free) with `tips_alloc` (and `tips_free`). Below we discuss how we annotated the NVMM stores with `tips_ulog_add`.

**Index with Single Pointer Updates.** Indexes for which write operations involve updating only a single pointer such as a Hash Table (HT), BST, and CLHT requires a `tips_ulog_add` before updating the HT/BST/CLHT node in the insert and delete logic, similar to the example shown in Figure 3. The lock-free indexes (LFHT/LFBST) use atomic Compare and Swap (CAS) and we added a `tips_ulog_add` before the CAS logic such that `tips_ulog_add` will be called again upon a CAS retry. Note that repeated UNDO logging will neither impact the correctness nor the performance as TIPS performs UNDO logging for an address only at the time of its first modification (§4.3.2). All such conversions require only 5-8 lines of change to the existing volatile codebase (Table 2).

**Index with Multi-Pointer Updates.** For the indexes like the B+tree or ART `tips_ulog_add` is added to backup the B+Tree/ART node before the triggering node split/merge operation. Also, `tips_ulog_add` is added before modifying the B+Tree/ART node in the normal case insert and delete logic. Totally it required only 11 and 9 LoC changes in the B+tree and ART codebase respectively. Note that none of the indexes require any modification in their read and scan logic.

**Comparison with other Conversion Techniques.** PRONTO [50] requires developers to manually add `op_begin()` and `op_commit()`. With NVTraverse [21] developers must first modify the index implemetation to a "traversal" index and then manually add the `ensureReachable` and `makePersistent` APIs. Similar to the `tips_ulog_add` (§3.2.6), the aformentioned APIs will internally issue `p-barrier` without requiring any user intervention. Additionally, unlike PRONTO and NVTraverse, TIPS can be used on indexes supporting different concurrency models. Like TIPS, both PRONTO and NVTraverse formally prove that their conversion yields correct persistent algorithm by guaranteeing DL. As also reported in the NVTraverse paper, RECIPE [37] can not always guarantee a correct conversion, even for the indexes that fall under their prescribed condition.

Moreover, RECIPE does not formalize the guarantees of their conversions and does not discuss the implications of guaranteeing BDL. Their updated ArXiv version [38] prescribes to selectively add `p-barrier` to specific loads to guarantee DL. But it is left to the developers to figure out the loads that needs to be correctly flushed; this further complicates the conversion. Alternatively, TIPS requires only minimal and simple modifications in the volatile codebase. We also formally describe how our conversion guarantees DL and yields a correct persistent algorithm for all our conversions.

| YCSB Workload | Read-Write-Scan % | Workload Nature |
|---|---|---|
| A | 50-50-0 | Write Intensive |
| B | 95-5-0 | Read Intensive |
| C | 100-0-0 | Read Only |
| D | 95-5-0 | Read Latest |
| E | 0-5-95 | Short Range Scan |

**Table 1:** Characteristics of YCSB workloads.

| Indexes | Concurrency Control | LoC |
|---|---|---|
| Hash table (HT) | Readers-writer lock | 5/211 |
| Lock-Free HT (LFHT) [51] | Non-blocking reads and writes | 5/199 |
| Binary Search Tree (BST) | Readers-writer lock | 5/203 |
| Lock-Free BST (LFBST) [12] | Non-blocking reads and writes | 5/194 |
| B+tree | Readers-Writer lock | 8/711 |
| Adaptive Radix Tree (ART) [40] | Non-blocking reads/blocking writes | 9/1.5K |
| Cache-line Hash Table (CLHT) [16] | Non-blocking reads/blocking writes | 8/2.8K |
| Redis [8] | Blocking reads and writes | 18/10K |

**Table 2:** Lines of code (LoC) to convert volatile indexes using TIPS.

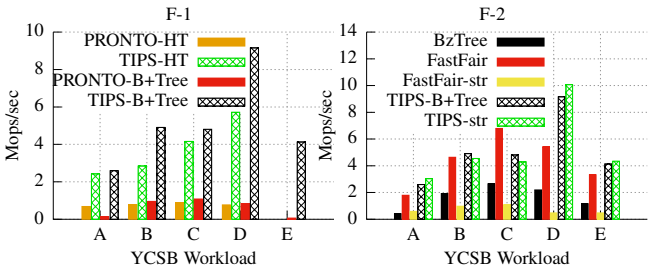


**Figure 4:** Performance comparison of TIPS against PRONTO for Hash Table (HT) and B+Tree (F-1) and TIPS-B+Tree against the NVMM-optimized B+Tree indexes– FastFair and BzTree (F-2).

### 7.2 TIPS vs. Other Conversion Techniques

**TIPS vs. PRONTO [50].** PRONTO is the state-of-the-art technique to convert globally blocking indexes with DL guarantee. As shown in Figure 4 (F1), both TIPS-HT and TIPS-B+Tree outperform the PRONTO counterparts by 20× across all workloads. Although both TIPS (TIPS-HT, TIPS-B+Tree) and PRONTO use RW lock for concurrency, TIPS can process the reads and writes concurrently in the DRAM-cache, and hence it shows a better performance. Also, PRONTO's overhead mainly comes from the synchronous waiting of writers for its background thread to complete the logging. Our performance profiling on the PRONTO-HT reveals that about 25% of the execution time is spent on synchronous waiting. In TIPS, there is no such synchronous waiting as it does not have any separate logging threads. Instead, writers will perform logging in their private OLog. Another source of overhead is the blocking during snapshots, which accounts for 8% of the execution time. Moreover, PRONTO builds its index on DRAM, so it can not scale beyond the DRAM capacity while TIPS can scale up to the NVMM capacity. This is a critical design benefit because legacy applications adopt NVMM not

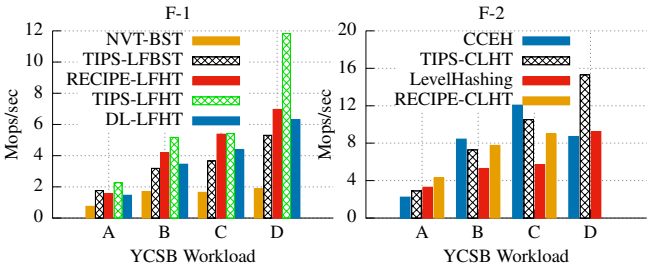


**Figure 5:** Performance comparison of TIPS with NVTraverse (F-1), RECIPE (F-1, F-2) and TIPS-CLHT with NVMM-optimized hash indexes– CCEH and LevelHashing (F-2).

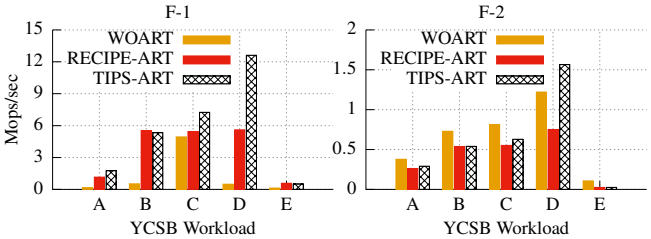


**Figure 6:** Performance comparison of TIPS-ART with RECIPE-ART and WOART for 32 threads (F-1) and 1 thread (F-2).

just for durability but also for its large in-memory capacity.

**TIPS vs. NVTraverse [21].** NVTraverse is the state-of-the-art technique to convert lock-free indexes with DL guarantee. As shown in Figure 5 (F-1), TIPS-LFBST outperforms NVT-BST by up to 3× across all workloads. Further analysis revealed that on average, each read and write in NVT-BST incurs 6 and 17 `p-barriers`, respectively, in the critical path. While TIPS-LFBST incurs only 2 `p-barriers` for each write in the critical path and reads never require `p-barrier`, thanks to the UNO logging and the DRAM-cache. Moreover, TIPS serves up to 25% of read requests from the DRAM-cache. So the readers do not need to traverse the BST on the NVMM for 25% of its read requests and hence a better performance.

**TIPS vs. RECIPE [37].** Comparing TIPS and RECIPE in terms of performance is not an apple-to-apple comparison as RECIPE supports only a weaker consistency (*i.e.*, BDL). Besides performance, we also stress how hard it is to achieve DL without trading off performance. Figure 5 and Figure 6 compare the performance of TIPS and RECIPE for LFHT, ART [39], and CLHT [16], respectively. TIPS indexes perform similar or better than RECIPE indexes across all workloads except for CLHT in workload A. This is because writes to CLHT incurs only one cacheline modification and one `p-barrier` in RECIPE. While in TIPS-CLHT, it incurs two `p-barriers` to commit the OLog. Nonetheless, TIPS-CLHT supports DL, and it performs mostly similar to NVMM-optimized hash indexes CCEH [53] and LevelHashing [66]. An easy way to guarantee DL, as proposed by Izraelevitz *et al.* [29] is to add a `p-barrier` for every reads and writes. We followed it to make a DL version of the RECIPE hash table (DL-LFHT in Figure 5). Such a conversion leads up to 1.4× drop in performance. One can also perform index-specific optimizations like the NVTraverse to guarantee DL. However, it requires expertise in NVMM programming and in-depth knowledge of the volatile index. Conversely, TIPS achieves DL with good performance and, notably, in an index-agnostic way.

### 7.3 TIPS vs. NVMM-optimized Indexes

**TIPS-B+tree.** Figure 4 (F-2) shows the performance of TIPS-B+tree against FastFair [26], and BzTree [11]. TIPS outperforms BzTree by up to 3× across all workloads; BzTree uses CoW and PMwCAS [59] to support crash consistency, and this generates a lot of NVMM write traffic. Unlike BzTree, TIPS-B+tree supports low overhead crash consistency using UNO logging and hence a better performance. TIPS performs similar or better than FastFair except for workload C. FastFair, with its smaller fanout (16), provides good point query performance, and TIPS-B+tree with a larger fanout (128) provides a good range query performance than FastFair. For string keys, FastFair (FastFair-str) performs up to 5× slower than TIPS (TIPS-str) as it loses its cache efficiency due to additional pointer chasing to retrieve string keys. Note that TIPS stores pointer to its keys for both string and integer keys; therefore no significant performance drop is observed.

**TIPS-ART.** In Figure 6, we present the performance of TIPS-ART and WOART [36]– volatile ART variant designed for NVMM. WOART is single-threaded, and hence we used a global lock for concurrency as done in previous work [37, 38]. WOART performs up to 40× slower than TIPS-ART across different workloads due to its poor concurrency model. For a single thread, TIPS-ART performs similar to WOART except for workload E. For workload E, the performance of TIPS-ART and RECIPE-ART are almost identical; this proves our claim in §4.1.6 that the additional OLog traversal in scan operations imposes negligible overhead. Our performance profiling revealed that only 2-3% of the time is spent on traversing OLog regardless of thread count.

### 7.4 Analysis on TIPS Design

**Write Scalability.** Figure 7 shows the scalability of the TIPS indexes. For write-intensive workload A, all TIPS indexes show good performance and scalability; for TIPS-B+Tree a sharp increase is observed after 16 threads. Because (1) for lower thread counts, the aggregate OLog size is less (OLog is per-thread). Moreover, as TIPS-B+Tree uses global RW lock TIPS backend writes are slower than the concurrent frontend writes, and consequently, foreground writers are blocked due to the lack of OLog space. (2) For higher thread counts, foreground blocking time is significantly reduced with a larger aggregate OLog size, Although the TIPS-HT uses RW lock, it is per-bucket and hence a better level of concurrency than the TIPS-B+tree. This enables TIPS backend to reply the OLog concurrently and hence a better performance for TIPS-HT. Still, for 16 threads, TIPS-B+tree outperforms PRONTO-B+Tree which also uses global RW lock by up to 3×.

**Read Scalability.** All TIPS indexes show good scalability for read-intensive workloads. Indexes supporting concurrent

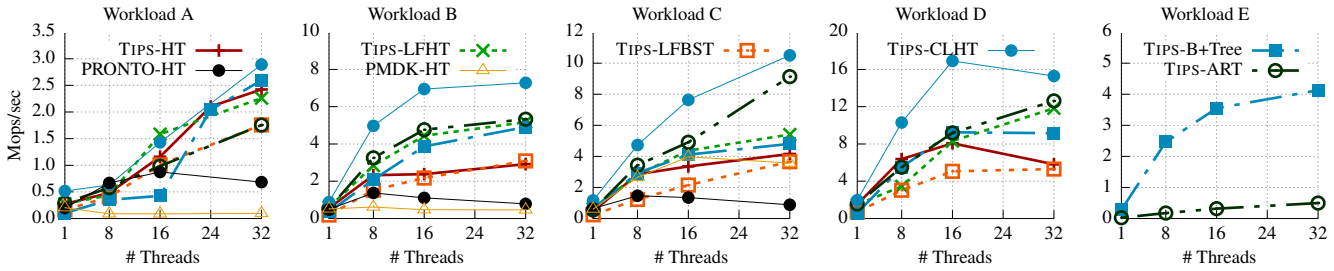


**Figure 7:** Scalability of TIPS-HT (RW lock), TIPS-B+Tree (RW lock), TIPS-LFHT, TIPS-LFBST, TIPS-CLHT and TIPS-ART.

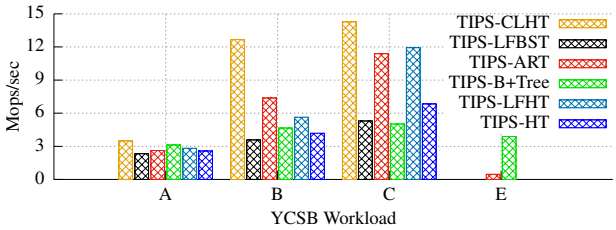


**Figure 8:** Performance of TIPS indexes for Zipfian workloads.

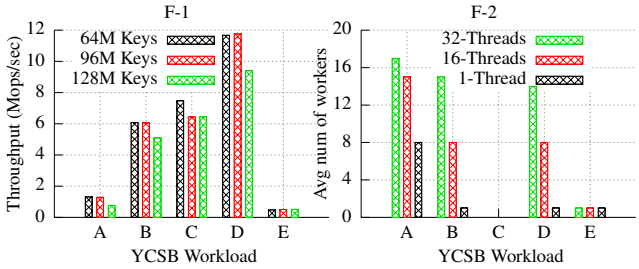


**Figure 9:** Studying the impact of large dataset (F-1) for 32 threads and adaptive scaling (F-2) for varying threads with TIPS-ART.

reads show good performance; for instance, TIPS-LFHT performs up to 1.2× better than TIPS-HT (RW lock) in workload C. All indexes regardless of their concurrency model shows high performance for workload D. Because workload D follows read-latest distribution, latest writes are being repeatedly read. Fortunately, the latest writes are most likely to be present in the DRAM-cache; hence a higher read hit ratio and consequently better performance. Although we cache only 25% of the keys for all workloads, the read hit for workload D is about 70%, while for other workloads it is less than 25%.

**Impact of UNO Logging.** Both TIPS-HT and PMDK-HT (Figure 7) use RW lock, and they both use the UNDO logging to guarantee crash consistency while updating the hash table. However, despite the similarities, TIPS-HT significantly outperforms PMDK-HT. The main performance bottleneck in PMDK is the logging operations in the critical path. This is evident from Figure 7, PMDK-HT performs and scales on par with TIPS-HT for workload C (100% reads), but its performance plateaus even for a small fraction of writes (5%) in workload B. With UNO logging, UNDO logging is kept off the critical path and consequently making TIPS scale better.

**Impact of Skewed Workloads.** Figure 8 shows the performance of TIPS indexes for Zipfian distribution; all indexes shows up to 2× better performance than the uniform distribution. Particularly for workload A, the chances of write coalescing in the ULog increases due to frequent updates to the same address. This reduces the number of UNDO logging performed, and further analysis revealed that the number of UNDO logging performed is reduced by 16% than that of uniform distribution. This further accelerates the background writes, and thus overall write performance is improved. For read-intensive workloads, repeated reading of hotkeys results in more DRAM-cache hits and eventually a better read performance. On average, the DRAM-cache hit ratio is increased by 5% for the Zipfian distribution. Overall, TIPS, in particular, UNO logging and DRAM-cache are well equipped to handle the skewed workloads. Note that we did not evaluate workload D as it supports read-latest distribution by default.

**Impact of Large Dataset.** Figure 9 (F-1) presents the performance of TIPS-ART for large datasets. While the performance for 64M and 96M keys is mostly the same across all the workloads, there is up to a 23% drop in performance for 128M keys. Particularly for workload C, the performance drop is also observed for 96M keys; for a larger dataset, the ART index grows bigger, and it results in increased pointer chasing to get to the leaf nodes, and hence there is a slight dip in the performance. Note that the RECIPE-ART also exhibits a performance drop of up to 16% across all workloads for 128M keys. Overall, this evaluation shows that TIPS as a system can handle much larger datasets effectively.

**Impact of Adaptive Scaling.** Figure 9 (F-2) shows the average number of background workers used for TIPS-ART. More workers are used for workload A as it is write-intensive, for all the other workloads workers count is relatively less as the write ratio is marginal. No workers are created for workload C as it is read-only. Workload E has the same write ratio (5%) as B and D, but TIPS employs only one worker for E. Because ART's scan is inherently slower, hence the foreground threads spend most of the time on the scan operation (*i.e.*, foreground writes are slow), this gives TIPS enough time propagate the updates. Thus our adaptive scaling can effectively adapt for the nature of workloads and the plugged-in index.

### 7.5 Sensitivity Analysis

**Sensitivity to DRAM-cache Size.** As shown in Figure 10, for read-intensive workloads, about 1.8-2.8× performance increase is observed as % keys cached increases. This is because the read hit ratio increases as more keys are being

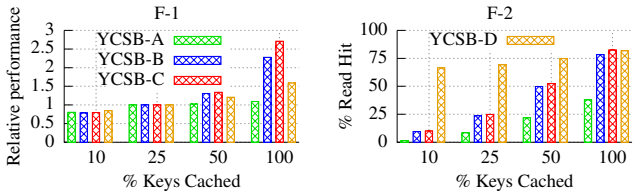


**Figure 10:** Performance sensitivity of TIPS-B+tree (F-1) and read hit % (F-2) for the varying DRAM-cache size. X-axis represents the % of keys cached in the DRAM-cache (default = 25%).

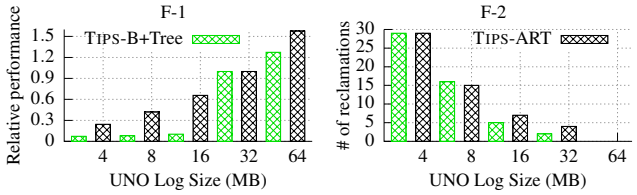


**Figure 11:** Performance sensitivity (F-1) and the number of log reclamations triggered (F-2) in TIPS-B+Tree and TIPS-ART for the varying UNO log size for Workload A (default = 32MB).

cached, enabling readers to complete their reads on the faster DRAM. Since workload A is write-intensive, it is less sensitive to DRAM-cache size. As more writes happen on the DRAM-cache, the applied keys are actively evicted, and it stores mostly the newly written keys. This poorly impacts the read hit ratio, and consequently, readers are forced to fall back to the B+tree on the NVMM.

**Sensitivity to UNO Log Size.** Figure 11 illustrates the performance of TIPS-ART, TIPS-B+tree for different UNO log sizes for the write-heavy workload A. As shown, there is a 4× and 9× performance drop for TIPS-ART and TIPS-B+tree with 32MB (default) and 4MB log size, respectively. This is because as the log size becomes smaller, the foreground writers are blocked for more time as TIPS-B+tree (RW lock) has a single-threaded backend. Whereas TIPS-ART supports concurrent backend and is relatively less affected by decreasing log size. Both TIPS-ART and TIPS-B+tree shows a 1.6× performance increase for 64MB because the log size is big enough to completely buffer all writes, and zero reclamations are triggered for 64MB. Also, note that more than 3 reclamations are triggered for a default log size of 32MB. The impact of smaller log sizes is relatively marginal for read-heavy workloads. The performance change is negligible until 8MB, and about a 20% performance drop is observed for 4MB log size.

### 7.6 Real-world Application: Redis

We ported a popular DRAM key-value store, Redis [8], using TIPS (TIPS-Redis). We compare its performance with the vanilla Redis running on the DRAM (DRAM-Redis) and NVMM (NVMM-Redis), and also with Intel's PMEM-Redis [7]. Note that NVMM-Redis does not ensure crash consistency and PMEM-Redis stores only the values in NVMM. Figure 12 shows the performance of Redis GET and SET operation evaluated using Redis-Benchmark [9]. We ran the benchmark for 32M keys (8-bytes) with a uniform random

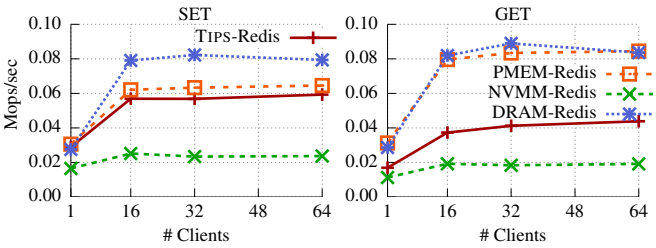


**Figure 12:** Performance comparison of TIPS-Redis with vanilla Redis running on DRAM and NVMM, and Intel's PMEM-Redis.

distribution. For SET operations, TIPS-Redis consistently outperforms the NVMM-Redis by 2.5×, and it performs up to 1.5× and 1.1× slower than the DRAM-Redis and PMEM-Redis, respectively. For the GET operations, TIPS-Redis perform up to 2× better than the NVMM-Redis and up to 2.2× slower than the DRAM-Redis and PMEM-Redis. TIPS-Redis maintains all the data and the Redis core on the NVMM, so (1) it provides a larger in-memory capacity (4× in our experiment) and immediate durability. (2) Both PMEM-Redis and DRAM-Redis take about 100 seconds to restore data from disk every time a server instance is created. While TIPS-Redis takes less than 1 second to recover upon safe termination.

### 7.7 Recovery

We performed the recovery test on all the TIPS-indexes. We injected crash 200 times arbitrarily using `SIGKILL`, similar to previous work [37, 41]. We also tested a crash during the recovery procedure. All TIPS-indexes successfully recovered after every crash. The worst-case recovery time would be a crash happening when the OLog is full. To measure this time, we injected a crash just when the OLog becomes full. Recovery time ranges between 0.5 and 9 seconds depending on the number of OLogs and concurrency control of the index.

## 8 Conclusion

We propose TIPS, a framework to systematically make volatile indexes and in-memory key-value stores persistent. At its core, TIPS adopts a novel DRAM-NVMM tiering to support index-agnostic conversion and durable linearizability. With the tiered concurrency model, TIPS achieves good scalability, performance, and enhanced applicability. UNO logging protocol is critical to achieve low crash consistency overhead and prevent persistent memory leaks. In our evaluation, we showed that TIPS could be effectively applied to indexes with varying concurrency models and the TIPS-enabled indexes shows excellent performance against the state-of-the-art index conversion techniques and NVMM-optimized indexes.

## Acknowledgement

We thank the anonymous reviewers and Haris Volos (our shepherd) for their insightful comments and feedback. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035).

## References


[1] Accelarating Redis with Intel Optane DC Persistent Memory. https://ci.spdk.io/download/2019-summit-prc/02_Presentation_13_Accelerating_Redis_with_Intel_Optane_DC_Persistent_Memory_Dennis.pdf.

[2] Aerospike Performance on Intel Optane Persistent Memory. https://www.aerospike.com/blog/performance-on-intel-optane-persistent_memory/.

[3] Bringing The Latest Persistent Memory Technology to Redis Enterprise. https://redislabs.com/blog/persistent-memory-and-redis-enterprise/.

[4] Intel Optane DC Persistent Memory NoSQL Performance Review. https://www.storagereview.com/review/intel-optane-dc-persistent-memory-nosql_performance-review.

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

[6] Optimize Redis With Next Gen NVM. https://www.snia.org/sites/default/files/SDC/2018/presentations/PM/Shu_Kevin_Optimize_Redis_with_NextGen_NVM.pdf.

[7] Pmem-Redis. https://github.com/pmem/pmem-redis/.

[8] Redis. https://github.com/antirez/redis.

[9] Redis Benchmark - How fast is Redis? https://redis.io/topics/benchmarks.

[10] Anandtech. Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here!, 2018. URL: https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb_apache-pass-is-here.

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

[12] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient Lock-Free Binary Search Trees. In *Proceedings of the 33th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, Paris, France, July 2014.

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

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

[15] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[16] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, March 2015.

[17] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. Poseidon: Safe, fast and scalable persistent memory allocator. In *Proceedings of the 21st*, Delft, Netherlands, December 2020.

[18] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD/PODS Conference*, pages 1243–1254, New York, USA, June 2013. ACM.

[19] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, January 2012. URL: http://doi.acm.org/10.1145/2094114.2094126, doi:10.1145/2094114.2094126.

[20] Keir Fraser. Practical Lock Freedom, 2004. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

[21] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, London, UK, June 2020.

[22] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Changwoo Min, and Dongyoon Lee. Witcher : Detecting crash consistency bugs in non-volatile memory programs, 2020. arXiv:2012.06086.

[23] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[24] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of Memory Reclamation for Lockless Synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007.

[25] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

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

[27] Intel. C++ bindings for libpmemobj (part 6) - transactions, 2016. URL: http://pmem.io/2016/05/25/cpp-07.html.

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

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

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

[31] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, pages 34:1–34:15, Porto, Portugal, April 2018. ACM.

[32] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*, pages 195–206, Hannover, Germany, April 2011.

[33] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. Mv-rlu: Scaling read-log-update with multi-versioning. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 779–792, Providence, RI, April 2019. ACM.

[34] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 468–479, Davis, CA, USA, December 2013.

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

[36] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February–March 2017.

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

[38] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. October 2019. arXiv:1909.13670v2 [cs.DC], https://arxiv.org/abs/1909.13670v2. arXiv:arXiv:1909.13670v2.

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

[40] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of Practical Synchronization. In *Proceedings of the International Workshop on*

*Data Management on New Hardware*, pages 3:1–3:8, San Francisco, California, June 2016.

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

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

[43] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

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

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

[46] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *Proceedings of the 17th Workshop on Hot Topics in Storage and File Systems*, Santa Clara, CA, July 2017.

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

[48] Paul E. McKenney. Structured deferral: Synchronization via procrastination. *ACM Queue*, pages 20:20–20:39, 1998.

[49] A. Memaripour and S. Swanson. Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software. In *Proceedings of the 36th International Conference on Computer Design*, Hartford, CT, October 2018.

[50] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[51] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, page 73–82, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/564870.564881.

[52] Micro. 3D XPoint Technology, 2019. URL: https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

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

[54] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dali: A Periodically Persistent Hash Map. In *Proceedings of the 31st International Conference on Distributed Computing (DISC)*, Vienna, Austria, October 2017.

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

[56] Ismail Oukid and Wolfgang Lehner. Data structure engineering for byte-addressable non-volatile memory. In *Proceedings of the 2017 ACM SIGMOD/PODS Conference*, Chicago, Illinois, USA, May 2017.

[57] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, Farmington, PA, November 2013. ACM.

[58] Li Wang, Zining Zhang, Bingsheng He, and Zhenjie Zhang. PA-Tree: Polled-Mode Asynchronous B+ Tree for NVMe. In *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE)*, Dallas, TX, April 2020.

[59] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy Lock-Free Indexing in Non-Volatile Memory.

In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 461–472, Paris, France, April 2018.

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

[61] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H.Alan Beadle, and Michael L. Scott. Interval-Based Memory Reclamation. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, wien, Austria, March 2018.

[62] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, London, UK, June 2020.

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

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

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

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