Making Volatile Index Structures Persistent using TIPS

R. Madhava Krishnan   Wook-Hee Kim   Hee Won Lee[*]
Minsung Jang[†]   *Sumit Monga   Ajit Mathew   Changwoo Min*
Virginia Tech   [*]Consultant   [†]Perspecta Labs

1 Introduction

Maturing and hardening an index is hard. Indexes are a fundamental building block in many storage systems. With the advent of Non-volatile Main Memory (NVMM), there have been a growing number of research efforts to develop new NVMM-optimized indexes. 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, (2) limited concurrent access, (3) not handling persistent memory leaks, and (4) not supporting variable-length keys. Such challenges in developing a new persistent index has led to a growing interest in porting the mature, volatile in-memory index to NVMM. However, prior works show that manually porting the legacy code for NVMM is complex requiring a lot of time and engineering effort and also it is error-prone [7, 8].

Limitations of the State of the Art. A few recent studies [2–4, 6, 9] proposed techniques to convert volatile indexes to NVMM, but they have some critical limitations. Most approaches have a limited applicability; NVTraverse [3] and link-and-persist [2] can be applied only to lock-free indexes; RECIPE [6] targets only lock-free or fine-grained lock based indexes and requires in-depth knowledge on a target volatile index for conversion; MOD [4] targets only purely functional data structures. PRONTO [9] builds the index on the DRAM preventing it from handling workloads beyond DRAM capacity. In addition, RECIPE [6] and MOD [4] support only a weaker non-durable linearizable consistency guarantee.

Our Approach. We propose **TIPS**, a systematic, index-agnostic conversion framework to convert a volatile index to NVMM supporting strong consistency guarantee.

2 Design of TIPS

Converting an Volatile Index using TIPS. TIPS APIs are classified into facade APIs and plug-in APIs, which are self-explanatory as shown in Figure 1. Developers can use the five facade APIs to access the plugged-in index and to plug-in a volatile index, they are required to modify their index implementation using TIPS plug-in APIs. As the example in Figure 2 suggests, applying TIPS to the existing volatile index follow this simple guideline; 1) replacing the volatile memory allocation (and free) with `tips_alloc` (and `tips_free`) and 2) intercept any modifications to the NVMM address space with `tips_ulog_add` call. The APIs and guidelines are same even for the complex indexes that requires multi-pointer updates (*e.g.*, B+tree split). Once an index is plugged-in to TIPS framework it becomes persistent.

```
/* 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 1: TIPS APIs.

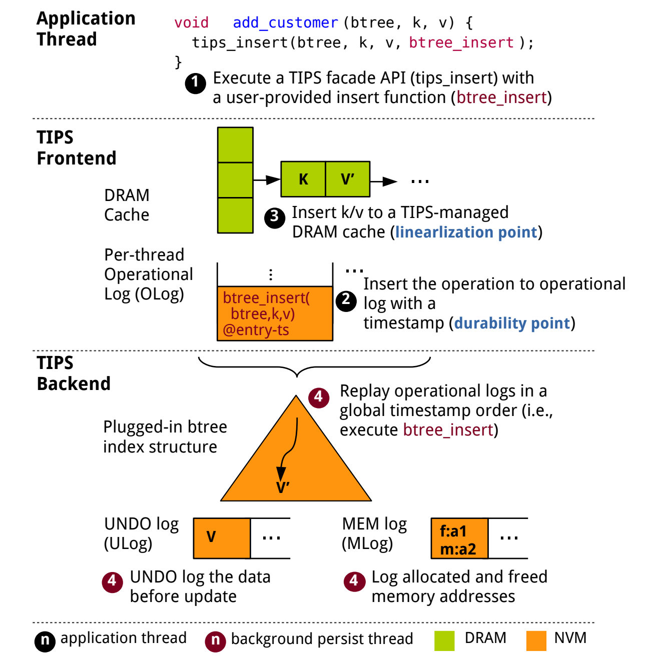
```
1    void hash_insert(hash_t *hash, key_t key, val_t value) {
2        node_t **pprev_next, *node, *new_node;
3        int bucket_idx;
4        pthread_rwlock_wrlock(&hash->lock);
5        // Find a node in a collision list
6        // Case 1: update an existing key
7        if (node->key == key) {
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 2: Code snippet of a TIPS-enabled hash table insert. TIPS plugin APIs are used for UNDO logging (Lines 9, 19) and memory allocation (Line 15).

Running Example. The Figure 3 gives an overview of how the access to the plugged-in index is managed within TIPS framework. When the user issues a write (❶) using one of the facade APIs, TIPS first records the operation in the per-thread operational log (OLog) to guarantee durability (❷) and then adds the key-value pair to DRAM-cache to make the writes visible to the readers (❸ is the linearlization point) and thus guaranteeing durable linearizability. TIPS-Backend then asynchronously propagates the writes to the plugged-in index by replaying the OLog entries in the global timestamp order (❹). Once the plugged-in index is updated the corresponding DRAM-cache entry is safely reclaimed. A lookup operation first looks for the requested key on the DRAM-cache and accesses the user plugged-in index on NVMM only when the DRAM-cache miss happens. A scan operation is always served from the NVMM index because it requires the full key-value data. Below we discuss the key insights of our design and briefly explain how they help TIPS to achieve its goals.

(1) DRAM-NVMM Tiering for Index-agnostic Conversion. The DRAM-cache provides an generic interface to access the plugged-in index. On a high level, DRAM-cache behaves similar to that of a CPU cache; it is transparent to the developers, and all the writes are absorbed in it. While plugging-in a volatile index, we essentially superimpose it on DRAM-cache; so the writes to the plugged-in index first goes to DRAM-cache

[*†]The authors contributed to this work while they were at AT&T Labs Research.



```
void  add_customer(btree, k, v) {
  tips_insert(btree, k, v, btree_insert);
}
```

Application Thread
❶ Execute a TIPS facade API (tips_insert) with a user-provided insert function (btree_insert)
TIPS Frontend
DRAM Cache
K
V'
...
❸ Insert k/v to a TIPS-managed DRAM cache (linearlization point)
Per-thread Operational Log (OLog)
btree_insert(btree,k,v) @entry-ts
❷ Insert the operation to operational log with a timestamp (durability point)
TIPS Backend
Plugged-in btree index structure
❹ Replay operational logs in a global timestamp order (i.e., execute btree_insert)
V'
UNDO log (ULog)
V
MEM log (MLog)
f:a1 m:a2
❹ UNDO log the data before update
❹ Log allocated and freed memory addresses
n application thread
n background persist thread
DRAM
NVM

Figure 3: Illustrative example of inserting a key-value pair in TIPS.

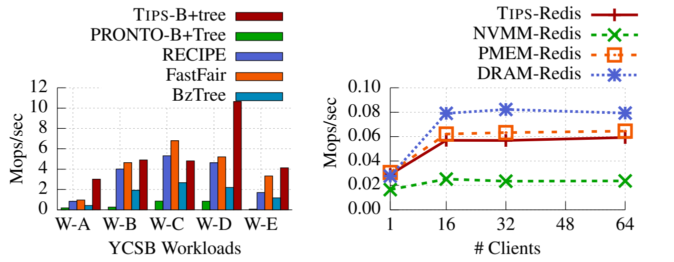


Figure 4: Performance comparison of TIPS-B+tree with RECIPE (P-BwTree), PRONTO, FastFair and BzTree (Left). Performance comparison of TIPS-Redis with vanilla Redis running on NVMM, DRAM and Intel's PMEM-Redis (Right).

and then the TIPS backend updates the plugged-in index in a crash consistent manner. Thus TIPS does not rely on any index-specific optimizations and does not place any restrictions on the concurrency model of indexes. This tiering between DRAM-cache and the plugged-in index enables an interesting concurrency model called the *tiered concurrency model* which is key for good performance and scalability of TIPS.

(2) Tiered Concurrency Model for Scalability. The DRAM-cache is concurrent hash table capable of supporting parallel readers and parallel disjoint writers. Hence, the requests that succeed in the TIPS frontend (all writes and read hits) will follow the concurrency model of DRAM-cache and the operations that go to TIPS backend (read misses and scan) will follow the concurrency model supported by the plugged-in index. This tiered concurrency model has excellent benefits in practice. It enables any volatile index regardless of its concurrency model to be plugged-in to TIPS without any restrictions unlike the previous studies [2, 3, 6]. Thus, even if an index using only a global mutex is plugged-in to the TIPS framework, all writes and reads hitting the DRAM-cache will be concurrently processed.

(3) UNO Logging for Index-agnostic Crash Consistency. It is important for TIPS to provide a generic crash consistency mechanism to support an index-agnostic conversion. At the same time, TIPS needs its crash consistency mechanism to be low overhead to ensure high performance and fast recovery. TIPS proposes UNO logging mechanism which synergistically uses the traditional UNDO logging (ULog) and operational logging (OLog) technique to support a low overhead durability and crash consistency; TIPS uses OLog to efficiently guarantee durability for the updates on the DRAM-cache (❷, avoiding costly UNDO logging in the critical path) and TIPS background thread uses the global ULog to ensure crash consistency while replying updates to the plugged-in index (❹, amortizing the UNDO logging cost with a larger batch). Finally, TIPS uses MLog to keep track of all the allocated/freed addresses to prevent persistent memory leaks across power cycles.

3 Evaluation

Evaluation Environment. We converted seven volatile indexes with different concurrency models and the Redis Key-value store using TIPS. All our index conversions required only 5-9 LOC changes in the existing codebase. Below we present the performance of TIPS enabled B+tree (TIPS-B+tree) and TIPS-Redis. We performed our evaluation on a 64-core Intel server equipped with a real NVMM using YCSB workloads with 32 Million integer keys. We set the size of the DRAM-cache to cache 25% of total keys.

B-trees. We evaluated TIPS against the state-of-the-art index conversion techniques RECIPE [6], PRONTO [9] and B+-tree indexes optimized for NVMM: FastFair [5]and BzTree [1]. Figure 4 shows that TIPS-B+tree outperforms all the state-of-the-art B+tree indexes. Apart from the performance, TIPS-B+tree guarantees strong consistency, supports variable-length keys and provides a fast and correct recovery internally addressing the persistent memory leaks all in an index-agnostic manner.

Redis Key-Value Store. Similarly, TIPS-Redis outperforms the vanilla Redis running on the NVMM (NVMM-Redis) and shows a comparable performance to vanilla Redis running on the fast DRAM (DRAM-Redis) and Intel's PMEM-Redis. Note that the PMEM-Redis stores entire Redis core and all keys on the DRAM and just stores the values on the NVMM. Apart from the performance, TIPS-Redis provides near instant recovery and 4× more capacity-scaling while the DRAM-Redis and PMEM-Redis can not scale beyond the size of the DRAM and it takes up to 100 seconds for both Redis versions to recover the data as they rely on the SSDs for durability.

References

[1] Arulraj *et al.* Bztree: A High-performance Latch-free Range Index for Non-volatile Memory, VLDB 2018.
[2] David*et al.* Log-free concurrent data structures, ATC 2018.
[3] Friedman *et al.* NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey, PLDI 2020.
[4] Haria *et al.* MOD: Minimally Ordered Durable Datastructures for Persistent Memory, ASPLOS 2020.
[5] Hwang *et al.* Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree., FAST 2018.
[6] Lee *et al.* RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes, SOSP 2019.
[7] Marathe *et al.* Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory, HotStorage 2017.
[8] Memaripour *et al.* Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software, ICCD 2018.
[9] Memaripour *et al.* Pronto: Easy and Fast Persistence for Volatile Data Structures, ASPLOS 2020.