

TL4x — Buffered Durable Transactions on Disk as Fast as in Memory

Gal Assa
Technion
Israel
galassa@campus.technion.ac.il

Andreia Correia
University of Neuchatel
Switzerland
andreia.veiga@unine.ch

Pedro Ramalhete
Cisco Systems
Switzerland
pramalhe@gmail.com

Valerio Schiavoni
University of Neuchatel
Switzerland
valerio.schiavoni@unine.ch

Pascal Felber
University of Neuchatel
Switzerland
pascal.felber@unine.ch

Abstract

The arrival of persistent memory devices to consumer market has revived the interest in transactional durable algorithms. Persistent memory (PM) is touted as having two attributes that distinguish it from other storage technologies: byte-addressability and fast transactional persistence.

In this work we investigate how these attributes differentiate PM from block storage in the context of buffered durability. We present a novel algorithm, TL4x, capable of providing buffered durable linearizable transactions with high scalability for disjoint writes and efficient persistence on either PM or block storage devices. TL4x is a software-only user-space solution that optimizes writes to persistent storage, providing buffered durable transactions whose cost is negligible compared to similar non-durable transactions. TL4x maintains a volatile consistent snapshot which is used for durability and shared with irrevocable read-only transactions, allowing long range-query operations to run in parallel with write transactions. We use TL4x to implement a transactional database engine that can outperform RocksDB by an order of magnitude.

1 Introduction

Persistent main memory has generated interest from academia and industry in efficient durable transactional algorithms. PM, also known as non-volatile main memory (NVMM), is being marketed as having two important advantages over block storage devices: PM has lower persistence latency and is byte-addressable. The first advantage is important when *durable* transactions are needed. Typically a transaction modifies a small amount of data, but subsequently induces a write to disk of at least one 4 kB page and issues at least two calls to `fsync()` [31, 43]. This approach is so slow when performed on disk (SSDs and HDDs) that no transactional

engine uses it by default. Instead, the majority of commercial and open-source database management systems (DBMS) provide *buffered* transactions by default when persisting to disk, *i.e.*, some committed transactions could be lost in case of a system crash. This indicates that a wide range of use cases can be satisfied with buffered durable transactions.

The second advantage, byte-addressability, allows access at a finer granularity and is considered beneficial from both programming and hardware perspective. While this is considered a PM-exclusive feature, we show that with buffered durable transactions, byte-addressability can be achieved by using DRAM backed by disk storage.

In this work, we present a novel persistent transactional memory (PTM) framework, *TL4x*. TL4x is a scalable, general purpose PTM which, when persisting to PM or to traditional block devices, induces negligible overhead compared to its non-durable performance.

TL4x’s design does not require write-ahead logging, which has been used to achieve durable transactions in DBMS for at least 30 years [38] (*e.g.*, redo-log for Microsoft Hekaton [10], MySQL InnoDB engine [39], PostgreSQL [45] and Oracle DB [33], or redo-log and shadow-copy for SAP HANA [13]). Instead, our technique maintains two volatile replicas of the data at all times, one of which could be frozen at any given time to contain a consistent snapshot of the data. This consistent replica is used by a background thread to copy the data to persistent storage, while read and write transactions continue to execute unhindered on the other replica. Regardless of the underlying non-volatile storage and the time it takes to persist data, with this design transactional execution is unaffected by the persist procedure: speculative transactions operate on the unfrozen volatile replica, and writers update the second volatile replica depending on TL4x’s state. Once the snapshot is no longer required, TL4x unfreezes its replica and applies the missed updates, while execution on the main replica continues uninterrupted. This snapshot update feature is the key to maintaining transaction performance unaffected by the persist procedure.

This is the author’s version of the work. It is posted here for your personal use. The definitive Version of Record was published in PPOPP23 <http://dx.doi.org/10.1145/3572848.3577495>, 2021.

TL4x makes additional use of the consistent replica to innately support snapshot reads in the form of *irrevocable read transactions*. These are useful when performing long range-queries on a data structure or a full table scan in a database, and also when a transaction performs an operation with external side effects, such as I/O requests. Traditionally, transactional memory frameworks accommodate only speculative transactions and do not support irrevocable ones, with range-queries often requiring extensive use of memory via multiversion concurrency control (MVCC), reaching up to 7× the memory usage in databases [5]. By definition, in-memory DBMS utilize at least 2× the size of the database, due to storing one replica of the database in DRAM and another in persistent storage. In practice this can grow even larger due to space amplification. TL4x has a constant memory overhead of 4× the size of the data and provides long range-query performance that is comparable to best performing volatile data structures.

The design of TL4x is agnostic about the underlying persistent storage device and, for the most part, so is its performance. It does not rely on the features of a specific persistent memory technology, such as the recently discontinued Intel Optane [25], and can work with any storage device, from legacy block devices to storage-class memory and persistent main memory.

With TL4x we make the following contributions:

- we introduce a highly scalable persistent transactional memory (PTM) with buffered durable linearizable transactions that can persist to PM or block storage (SSDs and HDDs);
- we eliminate the performance gap between persisting to PM, disk, or not persisting at all;
- we provide support for long read-only irrevocable transactions;
- we implement a fully transactional key/value store that surpasses the state of the art by 10× in throughput.

This paper is organized as follows. We first provide background and preliminaries in §2, and introduce the underlying system model in §3. We then describe the design of TL4x in §4. We experimentally evaluate its performance in §5, discuss related work in §6 and finally conclude in §7.

2 Background

Persistent main memory allows programmers to consider system crashes as normal events during the run of a program, by reducing the latency of a persistent write to a degree it could be performed in a synchronous manner. Its performance and benefits were thoroughly categorized and evaluated (e.g., Gugnani *et al.* [20] and Izraelevitz *et al.* [28]). Alas, persistence is neither trivial nor overhead-free and requires careful programming to secure the consistency of the persistent data in face of a crash. Programmers have therefore

to enforce order on writes to memory (*i.e.*, stores) in a way that allows recovery from a crash at any given time.

There exist multiple techniques for persisting data consistently. Write-ahead logging (WAL) is widely used by many PTMs and DBMS: changes are persisted to a log prior to applying them to the persistent data. Redo-logging logs the new data, so in case of a failure the changes can be re-applied. In undo-logging, the previous state of the data is logged and upon recovery the system rolls back to that state. Shadow-data is a technique that does not involve logging. It maintains more than one replica of the data and, upon successful update, a persistent pointer is toggled to point to the newest persistent replica.

The most popular definitions for the correctness of concurrent programs in presence of crashes are *durable linearizability* and *buffered durable linearizability*, as formalized by Izraelevitz *et al.* [26]. Durable linearizability extends linearizability [23] by requiring that real-time order with respect to crashes is preserved, *i.e.*, if an operation returned prior to a crash, its effects are visible after the crash. Buffered durable linearizability relaxes durable linearizability by requiring that the recovered state after a crash reflects *some* linearizable history, but not necessarily the most recent before the crash. Hence, it allows operations to be omitted from history, as long as causal order is preserved.

Transactional memory is a concurrency control concept in which multiple operations can be executed together safely. It is considered programmer friendly, as transactional memory frameworks handle concurrency on their own, and reduce programming effort by allowing the programmer to write sequential code. There exist both hardware and software implementations for transactional memory and this work presents a software transactional memory framework.

The design of TL4x relies on the TL2 algorithm [11]. In a nutshell, TL2 uses a shared global clock and every write transaction maintains two sets, read-set and write-set, to keep track of items read and written during the transaction, respectively. At the end of the speculative execution, these sets are used to validate the transaction’s commit-ability. We explain the process in detail in §4. If a transaction aborts due to inability to commit, or for another reason during its execution, none of its effects take place.

Unlike single operations spanning one object, there are multiple isolation levels (namely, consistency criteria) for transactional execution. This work focuses on linearizability, which means that there exists an equivalent sequential history in which every transaction takes place at a single point in time (the linearization point) and its effects are similar to those in the concurrent execution. The linearization imposes a total order on committed transactions. A handful of isolation levels for transactional systems exist, as described by [1, 4, 9], mostly differing in the states transactions are allowed to observe. More precisely, our proposed design satisfies a stronger notion than linearizability, *opacity* [19],

where even aborted transactions observe only consistent system states.

3 Model and Assumptions

In this section, we describe the system and failure models for which TL4x is designed and the guarantees it provides. We follow the model of Izraelevitz *et al.* [26], which assumes full system crash, and no spontaneous crash and recovery of individual processes.

TL4x assumes a hybrid memory architecture, with volatile caches and main memory, and additional persistent storage in the form of NVMM or disk. It is able to take advantage of both byte- and block-addressable non-volatile storage. Similar to [46], we use language interposition for load and store operations spanning persistent types during transactions.

For write ordering, we distinguish between disk and NVMM. In NVMM, we use cache-line write-back (CLWB) that is supported in Intel x86 architecture as persistent write-back, and store fence (sfence) for both *psync* and *pfence* in [26]. CLWB flushes a cacheline to persistent memory, and sfence orders CLWB and other store operations with respect to itself: store instructions after an sfence begin execution only after those before the sfence are complete. For disk, we use *msync()* to synchronize the write-back of mapped addresses to disk. TL4x uses *msync()* in a synchronous, *i.e.*, blocking, manner, so it corresponds to all of *pwb*, *psync* and *pfence* in [26].

TL4x guarantees *buffered durable linearizability* [26], *i.e.*, its transactions are linearizable and after a crash it is always possible to recover to a non-trivial state that represents a linearizable history. Unlike *durable linearizability*, not all committed transactions are required to be visible after a crash, only a consistent state. The amount of lost progress is not bounded by this correctness criteria nor by our design.

4 TL4x Design

In this section we describe the design and implementation of TL4x. We begin with a high level description of the algorithm and its different states, and then delineate the different building blocks. We conclude the section with a correctness proof sketch. For brevity of the algorithm’s presentation, we use continuous line numbering across TL4x’s sub-algorithms and mark thread-local variables with the `tl_` prefix.

4.1 High level description

TL4x uses four replicas of the user data to facilitate concurrency and persistence. Two replicas are volatile, *Main* and *Back*, and two persistent, P_0 and P_1 . All are memory mapped to four same-sized regions. There is a shared logical clock, in the form of an atomic counter. TL4x makes use of compare-and-swap and fetch-and-add for linearizability of transactions. We define a *Block* as a collection of consecutive memory addresses. TL4x uses an array of locks where each lock protects one block. In our implementation a block is 64

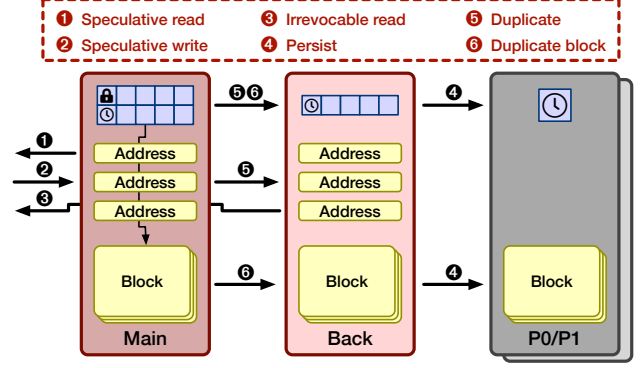


Figure 1. TL4x high level description.

bytes in size, occupying one cache line on x86. Every block on Main and Back is associated with an individual volatile *sequence* which corresponds to the value of the global clock on its last modification. Each of the two persistent replicas P_0 and P_1 has a single persistent sequence corresponding to it, indicating the timestamp of the latest modifications on any of its blocks.

Transactions operate on the volatile replicas Main and Back, and access the latter according to the system’s state. A background thread, the *copy thread*, manages the persistent copies and governs the system’s state transitions. The access types to each replica are described in Figure 1. TL4x provides three types of access, namely speculative read transactions, speculative write transactions and irrevocable read transactions. TL4x distinguishes between *speculative* and *irrevocable* read-only transactions: the former may abort and re-execute, whereas the latter are guaranteed to never abort.

TL4x cycles between four states that determine the access to the Back replica in the following order: DUPLICATE, SYNCHRONIZE, SNAPSHOT, and DUPLICATE_BLOCK. In DUPLICATE state, read and write speculative transactions operate on Main, as shown in Figure 1 (① and ②). Successful speculative writers apply modifications directly on Main during their execution and replicate those modifications to Back at commit time, updating both Main and Back’s block sequences (⑤). Irrevocable readers wait for the next time the system is in SNAPSHOT state and execute on Back where a consistent, linearizable snapshot is available. TL4x employs a user-space read-copy-update (URCU) mechanism to guarantee the consistency of Back. It is used to synchronize access to Back by speculative writers with TL4x’s state. We describe the URCU in detail below. After changing the state to SYNCHRONIZE, the copy-thread triggers a URCU synchronize (henceforth sync) and quiesces. Once sync begins, new write transactions do not update Back, until state is DUPLICATE or DUPLICATE_BLOCK. The copy-thread waits until all in-flight write transactions are complete and advances to SNAPSHOT.

During SNAPSHOT state, irrevocable readers read a frozen and consistent snapshot of Back, as shown in Figure 1 (③). During this state the sequences of Main are being updated to

the current commit clock, while the sequences on Back are left unchanged, thus causing the sequences in Main and Back to diverge. The copy-thread persists the snapshot to one of the persistent replicas (4). After the persist operation and irrevocable reads complete, the copy-thread transitions the state to `DUPLICATE_BLOCK`. In this state, write transactions copy every memory block they have locked from Main to Back and before unlocking it, update the corresponding sequences in Main and Back (6). Meanwhile, the copy-thread iterates over the used blocks: it acquires the lock, copies the block to Back, updates the sequence to match the one on Main, and releases the lock for each of the individual blocks whose sequence differs, thus aligning previously diverged sequences. When this phase completes, the copy-thread sets the state to `DUPLICATE`. The system will remain in this state until the next time a snapshot is required for an irrevocable read or persist.

The persistence mechanism toggles between the two persistent replicas and guarantees that at least one is consistent at any given time. Upon persist, the replica with the oldest sequence is chosen to be updated. Back is replicated into the chosen replica and then its sequence is set to the timestamp of the latest modification on any of Back's blocks.

4.2 Concurrency control

As its name implies, TL4x is based on *transactional locking*, and in particular employs eager locking by writers. Our design guarantees *opaque* transactions.

Speculative read and write transactions operate on Main, while irrevocable read transactions operate on Back. During write transactions, locks are acquired on the first attempt to write to an address in a block and released upon committing or aborting the transaction. All modified addresses are stored in a *write-set* along with their original values, to be used in case the transaction aborts. On speculative (read or write) transactions, read accesses are done with optimistic concurrency, without acquiring locks. Moreover, each memory location read during a write transaction is added to a *read-set*, to validate at commit time that no previously read address has changed. During irrevocable read transactions, read accesses are simply redirected to Back without the need for any atomic instruction.

We now describe the algorithms that implement TL4x concurrent durable transactions. In Algorithm 1 we present `beginTransaction()` and `endTx()`, which mark the start and end of the transaction, respectively. The user code must be placed between them. All loads and stores must be respectively interposed with `pload()` and `pstore()` presented in Algorithm 2, to accommodate our TL2 implementation. Although irrevocable transactions are not susceptible to conflicts, as they operate on the tranquil Back, all read accesses must be done through `pload()` to be redirected from Main to Back. When a conflict with another transaction is detected,

`abortTx()` is called and it triggers a restart to the beginning of the transaction.

The `beginTransaction()` function initializes a transaction depending on its type. The read- and write-sets are cleared (Line 7), and the global clock is sampled for the read-set validation (Line 12). Write transactions arrive on the URCU (Line 14), providing indication to the copy-thread that a write transaction may be updating Back. Irrevocable readers wait until Back is ready for a linearizable snapshot, namely, state is `SNAPSHOT` (Line 9). On `waitForSnapshot()` a reader increments the readers counter when the `COUNTER_ON` flag is set (Line 22). This implies that irrevocable read transactions may be starved, by never observing the `COUNTER_ON` flag set by the copy-thread on Line 126. But, in practice, there should be enough time for all pending irrevocable readers to increment the counter when the state is `DUPLICATE_BLOCK`. An irrevocable read can only start executing after the copy-thread transitions to state `SNAPSHOT` (Line 25). We show on §4.6 that when `waitForSnapshot()` returns, it guarantees that Back is consistent and that it will not be modified until the current transaction is done reading.

Read accesses for any transaction type are interposed with the `pload()` function to satisfy opacity, as shown in Algorithm 2. Irrevocable readers simply read the data from Back, without additional validation (Line 77). Speculative reads check that the desired address is not locked and was not modified after the timestamp sampled at `beginTransaction()` (Line 84). In this case the value can be returned to the caller and the execution may proceed. Otherwise the transaction aborts. Store interposition is done in `pstore()` also found in Algorithm 2. Writers acquire the locks relevant to a write and add the original value to the write-set. New values are written in-place to Main. Note that in our implementation some writes may span more than a block and therefore require the acquisition of multiple locks. Upon failure to acquire a lock, the transaction aborts.

After user code execution completes, the function `endTx()` in Algorithm 1 is called. The steps executed at the end of a transaction are defined by the transaction's type. Irrevocable readers are guaranteed to observe a linearizable snapshot and do not require validation before returning. When these transactions complete, the reader counter is decremented to enable further state transitions by the copy-thread. Speculative read-only transactions also do not require any commit-time validation, because if all read memory locations observe (Line 83) an associated lock version that precedes the sequence read at the start of the transaction (`tl_clock`, Line 12), then all read memory locations have the same values at the beginning of the transaction. Note that an update of a read memory location after `beginTransaction()` would result in the corresponding lock sequence being greater than or equal to `tl_clock`. Write transactions require validating that the locks protecting the read-set were not modified by a concurrent transaction until all the write-set locks were acquired.

Algorithm 1 — TL4x beginTx(), endTx() and abortTx()

```
1 // global variables
2 atomic<uint64_t> gClock; // global clock
3 atomic<uint64_t> gState {STATE_DUPLICATE}; // snapshot state
4 atomic<uint64_t> gCounter; // counter of pending readers

5 void beginTx() {
6     RETRY: // abortTx() jumps here
7     tl_writeSet.numEntries, tl_readSet.numEntries = 0;
8     if (tl_txType == TX_IS_IRR_READ) {
9         waitForSnapshot(); // wait until 'back' is a snapshot
10        return;
11    }
12    tl_clock = gClock.load();
13    if (tl_txType == TX_IS_UPDATE) {
14        urcu.arrive(tl_tid);
15        tl_state = gState.load();
16    }
17 }

18 void waitForSnapshot() {
19     while (true) {
20         uint64_t counter = gCounter.load();
21         if ((counter & COUNTER_ON) == COUNTER_ON) {
22             if (gCounter.cas(counter, counter + 1ULL)) { break; }
23         }
24     }
25     while (gState.load() != STATE_SNAPSHOT) {}
26 }

27 void abortTx() {
28     if (tl_tx_type == TX_IS_UPDATE) {
29         tl_writeSet.rollbackMain();
30         uint64_t nextClock = gClock.fetch_add(1) + 1;
31         if (tl_state == STATE_DUPLICATE_BLOCK) {
32             tl_writeSet.duplicateBlocksOnBack_UpdSeq(nextClock);
33         }
34         tl_writeSet.unlock(nextClock);
35         urcu.depart(tl_tid);
36     }
37     goto RETRY;
38 }

39 atomic<uint64_t> gLocks[NUM_LOCKS]; // array of locks
40 uint64_t gSeqMain[NUM_LOCKS]; // array sequences for 'main'
41 uint64_t gSeqBack[NUM_LOCKS]; // array sequences for 'back'

42 bool endTx() {
43     if (tl_txType == TX_IS_IRR_READ) {
44         gCounter.fetch_sub(1); // signal copy thread
45         return true;
46     }
47     // read-only transactions commit immediately
48     if (tl_writeSet.numEntries == 0) {
49         if (tl_txType == TX_IS_UPDATE) urcu.depart(tl_tid);
50         return true;
51     }
52     if (!tl_readSet.validate(tl_clock)) { abortTx(); }
53     uint64_t nextClock = gClock.fetch_add(1) + 1;
54     if (tl_state == STATE_DUPLICATE) {
55         // set new sequence on gSeqMain and gSeqBack
56         tl_writeSet.duplicateOnBack_UpdSeq(nextClock);
57     } else if (tl_state == STATE_DUPLICATE_BLOCK) {
58         // set new sequence on gSeqMain and gSeqBack
59         tl_writeSet.duplicateBlocksOnBack_UpdSeq(nextClock);
60     }
61     tl_writeSet.unlock(nextClock);
62     urcu.depart(tl_tid);
63     return true;
64 }

65 // duplicate modifications on 'back' and update sequences
66 void duplicateOnBack_UpdSeq(uint64_t nextClock) {
67     for (w : tl_writeSet) {
68         int idx = hidx(w.addr);
69         gSeqMain[idx] = nextClock;
70         memcpy(w.addr - BEGIN_MAIN + BEGIN_BACK, w.addr,
71             w.length);
72         gSeqBack[idx] = gSeqMain[idx];
73     }
74 }
```

Upon successful validation, the commit timestamp is obtained by incrementing the global clock. This timestamp will be used as the new sequence when releasing the write-set locks.

TL4x design relies on the Back replica being kept up to date. This implies that a thread that acquires a block's lock is responsible for duplicating its contents to Back. As such, before releasing the write-set locks, it may be necessary to copy modifications done on Main (or entire blocks) to Back, if the state is DUPLICATE, 56 (or DUPLICATE_BLOCK, Line 59). Copying entire blocks is required since it is possible that subsequent transactions modify multiple memory locations in the same block on Main during SNAPSHOT state, without reflecting the updates to Back. On both commit (Lines 49 and 62) and abort (Line 35), write transactions depart from the URCU.

Algorithm 1 shows the code for the transaction abort procedure upon conflict. During abortTx(), a transaction

releases the locks it acquired during execution and rolls back the changes to Main. In addition, it departs from the URCU, so the thread is no longer considered able to modify Back. As aborted transactions may have acquired locks during execution, they are held responsible for replicating the locked blocks to Back if they begin executing when the state is DUPLICATE_BLOCK (Line 32).

Memory management. TL4x follows a safe memory reclamation scheme and uses its own memory allocator for memory allocation and deallocation within transactions. It does not leak memory, neither volatile nor persistent, even in face of a crash, as its operations are considered part of transactions and are hence subject to the same all-or-nothing semantics. Roughly speaking, it maintains a collection of stacks that represent free blocks of different sizes. If a block of the required size is not available in the relevant stack, the

Algorithm 2 — Load and store interposition

```
74 template<typename T> struct persist {
75     uint64_t vrmain;
76     T pload() const {
77         if (tx_type == TX_IS_IRR_READ) {           // read from 'back'
78             T lval = *(T*)&vrmain - BEGIN_MAIN + BEGIN_BACK;
79             return lval;
80         }
81         T val = vrmain;
82         asm volatile ("": :: "memory");           // compiler fence
83         uint64_t sl = gLocks[hidx(&vrmain)].load(); // lock's state
84         if ((sl & LOCKED) || (sl > tl_clock)) { abortTx(); }
85         if (tl_type == WRITE_TX) { tl_readSet.add(this, sizeof(T)); }
86         return val;
87     }
88     void pstore(T newVal) {
89         int idx = hidx(&vrmain);
90         uint64_t sl = gLocks[idx].load();           // lock's state
91         if (sl != (LOCKED | tl_tid)) {
92             if (!(sl & LOCKED) && (sl <= tl_clock) &&
93                 gLocks[idx].CAS(sl, LOCKED | tl_tid)) {
94                 tl_writeSet.add(&vrmain);
95             } else { abortTx(); }
96         }
97         vrmain = newVal;
98     }
99 };
```

allocator allocates an additional block that meets the requirements. Upon freeing memory, the freed block is appended to the relevant stack.

4.3 Copy thread

The copy-thread serves two purposes: first, it transitions TL4x's state and second, it is responsible for copying data from Back to persistent storage. It runs on an infinite loop, described in Algorithm 3.

Initially, the state is DUPLICATE and the copy-thread determines if it is required to freeze a snapshot on Back. A snapshot is required if the global clock advances more than a predefined value (MIN_TXN_SYNC) since the last time Back was persisted, or when requested by an irrevocable reader, via the reader counter. We refer to these events as *persist event* and *read snapshot event*, respectively.

The copy-thread resets COUNTER_ON to prevent additional irrevocable readers from incrementing the readers counter. Then, it triggers a URCU sync, after which Back is a consistent snapshot of Main, at the time of the last committed or aborted transaction that did not observe the state as SYNCHRONIZE or SNAPSHOT.

If Back is frozen due to a persist event, the copy-thread first determines the persistent replica to update, by comparing the timestamps of p_0 and p_1 , and chooses the least updated. We denote the chosen replica as p . During `persistTo()`, the contents of Back are then copied to p in the following

manner. First, the timestamp of every block in Back is compared to the one recorded upon transition to SYNCHRONIZE and the timestamp of p . If it is between them, the block is copied from Back to p . Following this copy process, the relevant cache lines are flushed and a memory fence is issued. Next, the timestamp of p is updated, accompanied by flush, fence, or `msync()` for disk, making it safe to use for recovery.

Finally, it updates the timestamp of p . In both cases (persist event and read snapshot event), the copy-thread waits until all of in-flight irrevocable readers complete and then transitions the system into DUPLICATE_BLOCK state.

In DUPLICATE_BLOCK, the copy-thread sets COUNTER_ON to allow irrevocable readers to increment the reader counter. The copy-thread scans the lock and sequence arrays for blocks whose sequence on Main is higher than on Back and then locks and copies them to Back, skipping locked blocks. Before copying blocks, the copy-thread must trigger an additional URCU sync and quiesces, to guarantee that new writers observe the state as DUPLICATE_BLOCK. This is to ensure that every locked block is copied to Back. Otherwise, in-flight writers that observe a different state may unlock a block without updating Back, hence leaving it in an inconsistent state. Finally, the copy-thread transitions the state back to DUPLICATE.

When persisting to disk, `msync()` may fail [50]. Although not shown in Algorithm 3, an error in the `msync()` on Line 117 can be handled by repeating the call to `persistTo()` (Line 115) and retrying the `msync()`, while an error in the `msync()` of Line 122 can be handled by repeating the write to the timestamp (Line 118) before retrying the `msync()`.

The progress condition of the copy-thread is *blocking starvation-free* [22]. There are only three waiting loops, on Lines 124, 109 and 127, where the copy-thread waits for at most MAX_THREADS-1 other threads (irrevocable readers for the first, speculative writers for the others). The copy-thread uses COUNTER_ON to prevent irrevocable readers from incrementing the read counter, so it will be blocked by no more than MAX_THREADS-1 irrevocable-readers on Line 124. For writers, it is possible for the same number of threads to arrive on the URCU before the state changes to SYNCHRONIZE or DUPLICATE_BLOCK, on Line 109 or 127, respectively. `persistTo()` executes in parallel with irrevocable reads on Back, and cannot be blocked by speculative transactions. `copyMainToBack()` copies all blocks whose associated locks are available to be acquired by the copy-thread, and skips locked blocks, hence also not blocked.

4.4 Recovery procedure

Algorithm 4 details the recover mechanism behind TL4x. Recovery takes place upon initializing TL4x when there exists persistent data. It determines the persistent replica to read to be the more recent between P_0 and P_1 , henceforth denoted P_{cons} . We denote the second replica P_{incons} . The content of

Algorithm 3 – Copy thread loop

```
99 void copyThreadLoop() {
100   uint64_t lastSyncedTxn = 0;
101   while (!destructorCalled) {
102     bool doPersist = false;
103     if (gClock.load() < (lastSyncedTxn + MIN_TXN_SYNC)) {
104       if (gCounter.load() == COUNTER_ON) { continue; }
105     } else { doPersist = true; }
106     if (doPersist) { lastSyncedTxn = gClock.load(); }

107     gCounter.fetch_sub(COUNTER_ON);
108     gState.store(STATE_SYNCHRONIZE);
109     urcu.synchronize();
110     gState.store(STATE_SNAPSHOT);

111     if (doPersist) {
112       PMeta* p0 = (PMeta*)BEGIN_P0;
113       PMeta* p1 = (PMeta*)BEGIN_P1;
114       PMeta* p = (p1->ts > p0->ts) ? p0 : p1; // update older region
115       uint64_t usedSize = persistTo(p, lastSyncedTxn);
116       PFENCE(); // NVMM only
117       MSYNC(p + sizeof(p->ts), usedSize); // disk only
118       p->ts = lastSyncedTxn;
119       // the timestamp is persisted only after the user data is durable
120       PWB(&p->p_ts); // NVMM only
121       PSYNC(); // NVMM only
122       MSYNC(p, sizeof(p->ts)); // disk only
123     }
124     while (gCounter.load() > 0) { yield(); }
125     // wait for irrevocable reads
126     gState.store(STATE_DUPLICATE_BLOCK);
127     gCounter.store(COUNTER_ON);
128     urcu.synchronize();
129     copyMainToBack(); // update snapshot
130     gState.store(STATE_DUPLICATE);
131     doPersist = false;
132   }
}
```

P_{cons} is then copied to Main and Back without the timestamp. Then, P_{cons} is reflected without its timestamp to P_{incons} . The changes are flushed, and only afterwards the timestamp is updated and flushed. Finally, the global clock is set to the timestamp of $P_{cons}+1$.

4.5 User-space RCU

We implement a user-space RCU synchronization mechanism to be used by the copy-thread and write transactions. The URCU is integrated with TL4x's global clock. Write transactions *arrive* before execution, by reading the global clock and announcing this clock value (Line 14), and *depart* at the end of the transaction by clearing the announced clock (Lines 35, 49, or 62). The copy-thread executes an `rcu_synchronize()` and *quiesces* by reading the global clock and waiting for all in-flight write transactions whose announced timestamps are lower than or equal to the timestamp it read from the global clock (Lines 109 and 127).

The URCU sync executed after the SYNCHRONIZE state is set (line 109), guarantees that all transactions starting after the urcu sync returns, will not to modify Back while

Algorithm 4 – Recovery

```
133 void recover() {
134   // copy from the consistent persistent region based on the timestamp
135   PMeta* p0 = (PMeta*)BEGIN_P0;
136   PMeta* p1 = (PMeta*)BEGIN_P1;
137   PMeta* pcons = (p1->ts > p0->ts) ? p1 : p0;
138   PMeta* pincons = (p1->ts > p0->ts) ? p0 : p1;
139   memcpy(BEGIN_BACK, pcons, REGION_SIZE); // restore 'back'
140   memcpy(BEGIN_MAIN, pcons, REGION_SIZE); // restore 'main'

141   // don't copy the timestamp, which is the first word in PMeta
142   memcpy((uint8_t*)pincons + sizeof(pincons->ts),
143          (uint8_t*)pcons + sizeof(pcons->ts), REGION_SIZE -
144          sizeof(pcons->ts));
145   flushPWB(pincons, REGION_SIZE); // flushes NVMM cachelines
146   PFENCE(); // NVMM only
147   MSYNC(pincons + sizeof(pincons->ts), REGION_SIZE -
148          sizeof(pcons->ts)); // disk only
149   pincons->ts = pcons->ts;
150   PWB(&pincons->p_ts); // NVMM only
151   PSYNC(); // NVMM only
152   MSYNC(pincons, sizeof(pincons->ts)); // disk only
153   gClock.store(pcons->ts + 1);
154 }
}
```

the state is in SYNCHRONIZE or SNAPSHOT. Modifications on Back are made on lines 32, 56, and 59, these functions are executed iff the transaction has seen states DUPLICATE or DUPLICATE_BLOCK. Also, the URCU sync will return only after the completion of all write transactions which started before its invocation, guaranteeing that on-going transactions currently modifying Back have completed. The sync on DUPLICATE_BLOCK secures that writers will copy to Back every block the copy-thread may observe as locked during this state, before the next time TL4x enters SNAPSHOT. Given that the global clock is always incremented at the end of a committed write transaction (Line 53), it is impossible for a write transaction to depart and arrive again using the same timestamp, hence `urcu.synchronize()` is starvation-free.

4.6 Buffered durable linearizability

To show TL4x is buffered durable linearizable, we begin by showing that it is linearizable in the absence of crashes. The linearizability of speculative reads and writes is inherited directly from the TL2 algorithm, as from the perspective of Main, these are the only types of transactions. In a nutshell, versions of read addresses are validated prior to returning the read value during load interposition, so reads are opaque. For write transactions, items in the write-set are locked and at the end of the speculative execution the read-set items are validated again. If validation succeeds, the linearization point will be any point in time between the last lock acquisition of the write-set and the first released lock. For read-only transactions that do not abort during speculative execution, the linearization point is when the global clock is read in `beginTx()` (Line 12).

Irrevocable readers operate on Back and read the snapshot after the state is set to SNAPSHOT, which happens once the first RCU (Line 109) synchronize is complete. We now show that the observed snapshot is consistent, and reflects the linearizable history of Main up to the time the last transaction to complete among those which observe DUPLICATE or DUPLICATE_BLOCK states before RCU sync begins, which is also the linearization point for irrevocable reads. A transaction that completed before transition to SNAPSHOT could not be omitted: its effects are reflected to Back before it completes, since it observed one of DUPLICATE and DUPLICATE_BLOCK states upon `beginTx()`. The Snapshot will be read only after the changes are applied thanks to the RCU sync. Additional changes cannot be applied since transactions observing states SYNCHRONIZE or SNAPSHOT on `beginTx()` do not update Back. The next update to Back happens no earlier than during the next DUPLICATE_BLOCK state, but TL4x transitions to that state only after all snapshot reads are complete, namely, after the reader counter reaches 0 and the persist process, if invoked, completes.

DUPLICATE_BLOCK state guarantees that the effects of transaction that commit during SYNCHRONIZE or SNAPSHOT are reflected to Back, either by the copy-thread or by writer threads. Here, too, a transaction cannot be omitted: the copy-thread traverses blocks only after the RCU sync (Line 127) completes, hence all ongoing write transactions observed the DUPLICATE_BLOCK state. TL4x's state turns to DUPLICATE only after the copy-thread completes scanning the array of locks. Therefore, every block is either not modified during SYNCHRONIZE or SNAPSHOT, copied to Back by the copy-thread, or locked by a writing thread, which will reflect the content to Back upon unlocking.

Buffered durable linearizability stems from the consistency of the snapshot. The persist operation is in fact a snapshot read, observing the same snapshot as the irrevocable readers, which represents a linearizable history.

5 Evaluation

We evaluate TL4x and compare its performance to state of the art persistent transactional memory frameworks, key-value stores, and volatile data structures specialized for range queries. We use the Yahoo cloud service benchmark (YCSB) and DB-bench (from RocksDB) for comparing TL4x with other PTMs and key-value stores, and benchmark introduced by Arbel-Raviv and Brown [2, 41] for range-queries.

System settings. TL4x is implemented in C++ (compiled with gcc 10.3 and `-O2` flag for optimization), as are all other compared solutions. All experiments are performed on two different machines. The first machine is a two-socket server with an Intel Xeon Gold 5215 (2.5 GHz), 2× 128 GB 1st-gen Intel Optane DCPMM per socket (20 cores and 40 hardware threads in total), as well as 2×480 GB SATA SSDs. The second is a single-socket server with an Intel Xeon Gold 6326

(2.9 GHz) equipped with 2×128 GB 2nd-gen Intel Optane DCPMM, with 16 cores and 32 HW threads. It is equipped with a 512 GB NVMe SSD. Both servers run Ubuntu LTS, 20.04 and 22.04, respectively, and are configured to work in App-Direct mode for NVMMs. We observe similar trends on both machines, and show the results obtained on the first.

5.1 Workloads and compared systems

The range-query (RQ) benchmark [2] instantiates a data-structure and concurrently runs operations on it according to configured key-range, operation type ratio, and thread-count. We use it in two different experiments. The first experiment evaluates performance and scalability as RQ size increases, whereas the second fixes RQ size and measures throughput as the number of RQ threads increases.

We compare TL4x against a variety of volatile data structures that are specialized for range-queries, in addition to TrinityVRTL2 PTM [46]. We apply our PTM to a skip list, a relaxed AVL tree (RAVL), and a zip tree (sequential data structures) and compare them to a skip list, binary search tree, and a citrus tree (concurrent data structures), which have been enhanced to support range queries using techniques like versioned CAS (vCAS) [55] and bundled references [41]. These two techniques make use of multi-versioning to support linearizable snapshot reads. In the bundled references approach, RQs traverse the object version-history, where every update creates a new version. In vCAS, snapshot reads also traverse the version history, but new versions are created in response to invocations of snapshot reads.

YCSB and db_bench are standard benchmarks for databases and key-value stores. They include a handful of workloads that simulate real-world use-cases. We run YCSB workloads A and B. YCSB A is a write intensive workload, where every thread executes 50% reads and 50% updates. YCSB B is read dominant, and executes 95% reads and 5% updates. In addition, we run the following db_bench workloads: *fillseq* (writes keys in sequential order), *fillrandom* (writes keys in random order), *overwrite* (updates values for existing keys in random order), *readrandom* (reads keys in random order), and *readwhilewriting* (read keys in random order while a dedicated writer thread executes). Values are 100 bytes and keys are 16 bytes in size. We compare TL4x against the following key-value stores: TrinityDB [46]—based on TrinityVRTL2 PTM, a durably linearizable transactional memory framework which maintains a volatile replica to improve the efficiency of reads, RocksDB [37]—an in-memory key-value store that persists updates to disk or PM, either synchronously or asynchronously, and pmemkv [44]—the key-value store supplied with the persistent memory development kit (PMDK). For TL4x, we run all experiments for both PM and disk persistence, and define MIN_TXN_SYNC as 10 K transactions. For TL4x, we also run all experiments on a version that does not persist data at all (namely, the

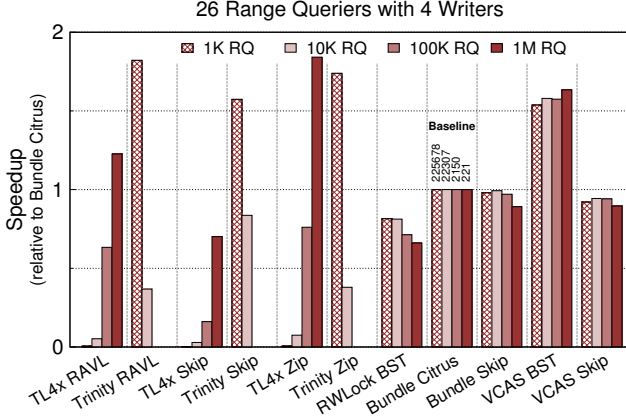


Figure 2. RQ benchmark with varying range query sizes.

code between Line 111 and Line 123 of Algorithm 3 is not executed).

5.2 Experimental results

We begin with evaluating range-queries in the form of irrevocable snapshot reads. We limit the key range to 1 M (10^6), and define dedicated threads for RQs and for updates. Each updating thread performs 50% inserts and 50% delete operations, to ensure the data structure is changing but maintains its size. Each experiment runs for 60 seconds. We measure update throughput and RQ throughput separately. All presented results for TL4x refer to disk persistency when not explicitly written in the label.

Figure 2 describes the results for the range-query micro-benchmark in the following setting: the thread-count is fixed at 30, where 4 threads constantly update the data structures, and 26 run range-queries. We experiment with multiple fixed RQ sizes. For each size, we measure the RQ throughput. The results are displayed as the relative speedup compared to a citrus tree with bundled references. The actual throughput of the citrus tree in RQs per second is noted on top of the bars associated with it.

We observe that TL4x becomes more competitive with the optimized data structure as RQ size increases. For small RQs TL4x performs worse due to the overhead during the synchronization phase outweighing the time it takes to perform the actual query. This cost becomes less noticeable for longer RQs. For large RQ sizes TL4x becomes more competitive, thanks to the irrevocable read transactions that operate on a single snapshot, and neither traverse multiple versions nor synchronize the read accesses with other threads.

The vCAS BST performs best overall for all RQ sizes, except for 1K queries where it is outperformed by TrinityVRTL2. Trinity benefits from low overhead when reading from its volatile replica and low abort rate for small query sizes. As following experiments show (YCSB B in particular), if short RQs were executed speculatively on TL4x and not as an irrevocable transaction, the performance would

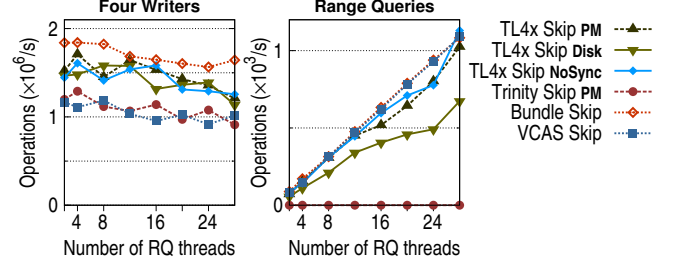


Figure 3. Range-query benchmark for skip lists.

match Trinity’s, as in both PTMs reads operate on a volatile replica, and both use TL2 for speculative transactions. On the other end of the scale, Trinity is unable to complete large RQ operations due write-read conflicts causing restarts.

Next, we fix the range-query size to 200 K, and the number of update threads to 4. We gradually increase the number of range-query threads, and measure update and range-query throughput. The results are shown in Figure 3 for skip lists and Figure 4 for search trees. For clarity of the presentation, Figure 4 shows only the results of TL4x with Disk persistence.

The update throughput of TL4x is on par or better, with the experimented range-query optimized data structures. For range-queries, TL4x also shows throughput that is comparable or better than the volatile data structures. Here, TrinityVRTL2 suffers from contention as in the previous experiment, and is unable to complete large queries. Notice the gap in RQ throughput between disk and PM skip lists on Figure 3, whose root cause is that RQ threads wait for the copy-thread’s state transitions. In turn, the copy-thread, awaits the completion of the synchronous `msync()`, which is subject to the higher latency of disk-writes. However, there is no gap between persisting to PM and not persisting.

It is important to emphasize that TL4x’s performance is comparable to the volatile data structures, in spite of providing transactional semantics and persistence. In addition, TL4x does not require the programming effort and expertise as vCAS and bundled references, which also impose applying the techniques to each data structure separately.

Regarding the KV-store experiments, Figure 5 shows the results of YCSB workloads A and B. We measure the throughput versus the number of working threads. For YCSB A, we see that TL4xDB dominates the other key-value stores. This is due to the usage of the TL2 concurrency control, combined with buffered writes, *i.e.*, write transactions may return before their effects are persisted. For YCSB B, TL4x achieves results that are similar to TrinityDB, and both outperform the alternatives. The similarity is unsurprising, seen as TL4x and TrinityVRTL2 share a similar concurrency control for speculative transactions, namely TL2, and both make use of volatile replicas of the data. In this benchmark RocksDB is executed without the `-sync` flag, *i.e.*, in a buffered manner.

Finally, we run a set of workloads from the `db_bench` suite, as detailed previously. The results are depicted in Figure 6. We examine four different configurations for RocksDB: with

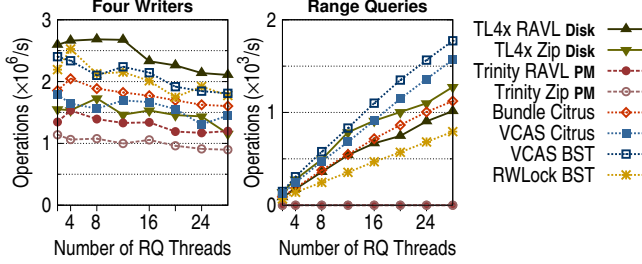


Figure 4. Range-query benchmark for search trees.

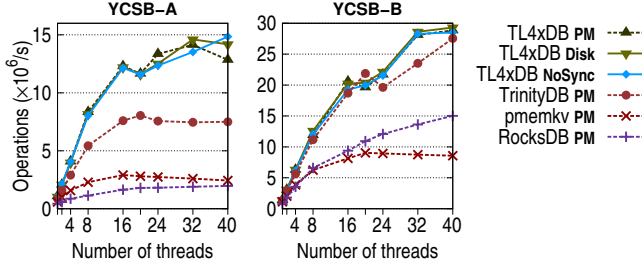


Figure 5. Write intensive (YCSB-A) and read dominant workloads (YCSB-B), on DBs with 10^6 keys, values of 100 bytes.

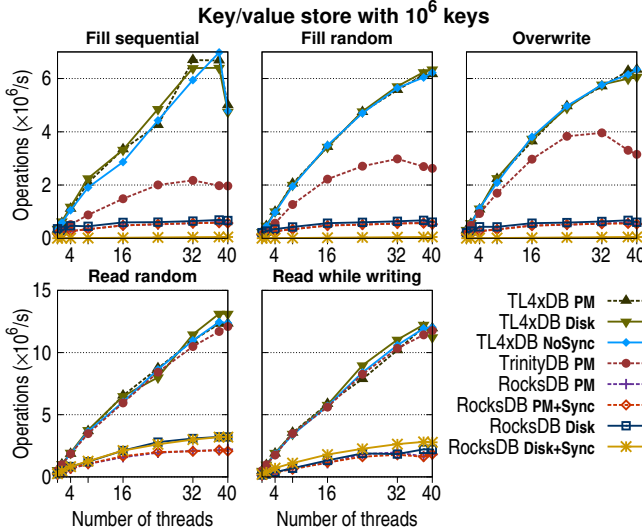


Figure 6. db_bench benchmark with 10^6 keys.

and without the `-sync` flag, and persisting to PM or to disk. For RocksDB, operations are not placed inside a transaction as this new feature is still unstable. Again, TL4xDB outperforms all alternatives in the write-intensive workloads and is matched only by TrinityDB in the read-mostly workloads, for the same reasons as in YCSB B. Specifically, TL4xDB Disk relative performance over RocksDB Disk is of 10x for *fillrandom* and 4x for *readrandom*.

TL4x does not utilize a WAL, and thus provides two benefits. First, it avoids log serialization, unlike RocksDB where all write operations must log their key/value, which effectively serializes them. Second, persist events in TL4x can be triggered synchronously (on-demand) by users through a blocking API, instead of asynchronously (every fixed write

transactions, via `MIN_TXN_SYNC`). Although persistence can be triggered synchronously, users have no control over how many transactions can execute until persistence is complete, *i.e.*, there exists no bound on the number of lost transactions upon system crash. This lack of guarantee is inherent in any design which does not serialize write transactions to help persistence take place. RocksDB can also loose any number of transactions in the case of a crash [12].

TL4x requires 4x storage usage. This drawback is attenuated by the fact that, unless using compression, any WAL-based in-memory DBMS must occupy at least one replica of the data in memory, another on persistent storage, and at least one persistent transactional log, which can grow to the size of the entire DB (times log amplification) if transactions modify a large fraction of the database. RocksDB relies heavily on data compression to reduce data usage, which induces a performance overhead.

TL4x suffers from a significant performance drop for 40 worker threads in the *fillseq* workload. This is due to over-subscription, when thread count (including copy-thread) exceeds the number of HW threads. The nature of this workload, which adds keys to the KV-store sequentially, causes a thread that is preempted after obtaining a lock to penalize the entire system and prevent its progress.

Disk persistence versus PM persistence. Throughout all experiments, TL4x’s performance for disk persistence, PM, and no persistence is similar or comparable, across two generations of Intel Optane PMM and two SSD types. This is thanks to the copy-thread persisting Back in a buffered fashion in the background, allowing concurrent transactions to operate uninterfered on volatile memory.

The in-memory snapshot, inherent to TL4x’s persistence approach, allows for the successful execution of long read-only transactions with irrevocable semantics, which prevents starvation caused by write transactions when those read queries are executed speculatively.

6 Related Work

libpmemobj is a library that provides durable transactions, but without concurrency. pmemkv is a key/value that provides basic key/value operations with some of its engines supporting concurrent operations. Both are part of Intel PMDK [44], and do not support block storage. There exist other PTM libraries designed for NVMM such as NVHeaps [7], Mnemosyne [53], Romulus [8], and OneFile [47].

Kamino-Tx [35] is a PTM with two data replicas which provides buffered durable linearizable transactions. Unlike TL4x, it is object-based and utilizes an undo log for durability. TimeStone [48] is another object-based PTM that employs a logical redo-log, and MVCC for read-only snapshots. Object-based PTMs/STMs impose a particular object-aware API for every access to an object, deviating from the classical STM approach of letting developers write generic sequential code.

Many software-based techniques for NVMM programming use a redo-log [17, 18, 24, 32, 34, 36, 53, 56], or an undo-log [6, 7, 30, 56, 57]. TL4x does not use logging.

Significant research effort focuses solely on creating or adapting existing data structures to work on NVMM, with examples being [3, 14–16, 21, 27, 29, 40, 42, 49, 51, 52, 54, 58]. These techniques do not provide transactional consistency, making them hard to apply to records of a database, nor are they designed for block storage devices.

7 Conclusion

For the time being, in-memory DBMS rely on buffered durability to provide acceptable performance to the end-users. The design of TL4x embraces this durability relaxation to create a PTM which outperforms other fully durable (non-buffered) state of the art PTMs like TrinityVRTL2.

With buffered durable linearizability, TL4x diminishes the performance gap between NVMM and block devices. Thus, a variety of applications can run on disk instead of persistent main memory, when strict durability is not required. The same could hold for other buffered PTMs. When compared with RocksDB, our TL4x in-memory DBMS achieves an order of magnitude better throughput for high thread count.

The TL4x PTM is portable, and will become available as open-source upon publication.

References

- [1] Atul Adya. 1999. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. Ph. D. Dissertation. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science.
- [2] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing Epoch-Based Reclamation for Efficient Range Queries. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) (PPoPP '18). Association for Computing Machinery, New York, NY, USA, 14–27. <https://doi.org/10.1145/3178487.3178489>
- [3] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. 2022. Detectable recovery of lock-free data structures. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 262–277. <https://doi.org/10.1145/3503221.3508444>
- [4] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’Neil, and Patrick O’Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.
- [5] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *Proc. VLDB Endow.* 13, 2 (2019), 128–141. <https://doi.org/10.14778/3364324.3364328>
- [6] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices* 49, 10 (2014), 433–452.
- [7] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices* 46, 3 (2011), 105–118.
- [8] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th Symposium on Parallelism in Algorithms and Architectures*. ACM, 271–282.
- [9] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is believing: A client-centric specification of database isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 73–82.
- [10] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server’s memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1243–1254.
- [11] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *DISC*. Springer.
- [12] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruva Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *CIDR*, Vol. 3. 3.
- [13] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. *ACM Sigmod Record* 40, 4 (2012), 45–51.
- [14] Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleftherios Kosmas. 2022. The performance power of software combining in persistence. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 337–352. <https://doi.org/10.1145/3503221.3508426>
- [15] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: in NVRAM data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 377–392. <https://doi.org/10.1145/3385412.3386031>
- [16] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: making lock-free data structures persistent. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1218–1232. <https://doi.org/10.1145/3453483.3454105>
- [17] Ellis R. Giles, Kshitij Doshi, and Peter Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14. <https://doi.org/10.1109/MSST.2015.7208276>
- [18] Jinyu Gu, Qianqian Yu, Xiyang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (Renton, WA, USA) (USENIX ATC '19)*. USENIX Association, USA, 913–928.
- [19] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, Siddhartha Chatterjee and Michael L. Scott (Eds.). ACM, 175–184. <https://doi.org/10.1145/1345206.1345233>
- [20] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.* 14, 4 (dec 2020), 626–639. <https://doi.org/10.14778/3436905.3436921>
- [21] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 775–788.

- <https://doi.org/10.1145/3373376.3378472>
- [22] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. <https://doi.org/10.1145/1146381.1146382>
- [23] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.
- [24] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) (*USENIX ATC '17*). USENIX Association, USA, 703–717.
- [25] Intel. 2022. Intel earnings statements, for Q2 2022. <https://download.intel.com/newsroom/2022/corporate/Intel-CEO-CFO-2Q22-earnings-statements.pdf>.
- [26] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*. Springer, 313–327.
- [27] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9888)*, Cyril Gavoille and David Ilcinkas (Eds.). Springer, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23
- [28] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir-saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).
- [29] Ana Khorguani, Thomas Ropars, and Noel De Palma. 2022. ResPCT: fast checkpointing in non-volatile memory for multi-threaded applications. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 525–540. <https://doi.org/10.1145/3492321.3519590>
- [30] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) (*ASPLOS '16*). Association for Computing Machinery, New York, NY, USA, 399–411. <https://doi.org/10.1145/2872362.2872381>
- [31] Linux. 2019. fsync() man page. <http://man7.org/linux/man-pages/man2/fdatasync.2.html>.
- [32] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 329–343.
- [33] Kevin Loney. 2004. *Oracle database 10g: the complete reference*. McGraw-Hill/Osborne London.
- [34] Youyou Lu, Jiwu Shu, and Long Sun. 2015. Blurred persistence in transactional persistent memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 1–13.
- [35] Amir-saman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnathan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) (*EuroSys '17*). Association for Computing Machinery, New York, NY, USA, 499–512. <https://doi.org/10.1145/3064176.3064215>
- [36] Amir-saman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) (*ASPLOS '20*). Association for Computing Machinery, New York, NY, USA, 789–806. <https://doi.org/10.1145/3373376.3378456>
- [37] Meta. 2017. RocksDB. <http://rocksdb.org/>.
- [38] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.
- [39] MySQL Developer Zone. 2019. MySQL 5.0 Reference Manual - Redo log. <https://dev.mysql.com/doc/refman/5.7/en/innodb-redo-log.html>.
- [40] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dali: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria (LIPIcs, Vol. 91)*, Andréa W. Richa (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 37:1–37:16. <https://doi.org/10.4230/LIPIcs.DISC.2017.37>
- [41] Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. 2022. Bundling Linked Data Structures for Linearizable Range Queries. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) (*PPoPP '22*). Association for Computing Machinery, New York, NY, USA, 368–384. <https://doi.org/10.1145/3503221.3508412>
- [42] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 371–386. <https://doi.org/10.1145/2882903.2915251>
- [43] Thanumalayan Sankaranarayanan Pillai, Vijay Chidambaram, Ramnathan Alagappan, Samer Al-Kiswani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 433–448. <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/pillai>
- [44] PMDK team. 2018. Persistent Memory Development Kit. <https://pmdk.io/pmdk/>.
- [45] PostgreSQL 9.0.23 Documentation. 2019. Write-Ahead Logging (WAL). <https://www.postgresql.org/docs/9.0/wal-intro.html>.
- [46] Pedro Ramalhete, Andreia Correia, and Pascal Felber. 2021. Efficient algorithms for persistent transactional memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–15.
- [47] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 151–163.
- [48] Madhava Krishnan Ramanathan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 335–349. <https://doi.org/10.1145/3373376.3378483>
- [49] Madhava Krishnan Ramanathan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff

- Kuenning (Eds.). USENIX Association, 773–787. <https://www.usenix.org/conference/atc21/presentation/krishnan>
- [50] Anthony Rebello, Yuvraj Patel, Ramnathan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. Can Applications Recover from fsync Failures?. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 753–767. <https://www.usenix.org/conference/atc20/presentation/rebello>
- [51] Anubhav Srivastava and Trevor Brown. 2022. Elimination (a, b)-trees with fast, durable updates. In *PPoPP ’22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 416–430. <https://doi.org/10.1145/3503221.3508441>
- [52] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, Gregory R. Ganger and John Wilkes (Eds.). USENIX, 61–75. <http://www.usenix.org/events/fast11/tech/techAbstracts.html#Venkataraman>
- [53] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.
- [54] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 93–111. <https://www.usenix.org/conference/osdi21/presentation/wang-qing>
- [55] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-Time Snapshots with Applications to Concurrent Data Structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) (PPoPP ’21). Association for Computing Machinery, New York, NY, USA, 31–46. <https://doi.org/10.1145/3437801.3441602>
- [56] Pantea Zardoshti, Tingzhe Zhou, Yujie Liu, and Michael Spear. 2019. Optimizing Persistent Memory Transactions. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 219–231.
- [57] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 897–912. <https://www.usenix.org/conference/atc19/presentation/zhang-lu>
- [58] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 128:1–128:26. <https://doi.org/10.1145/3360554>