Crafty: Efficient, HTM-Compatible Persistent Transactions

Kaan Genç
Ohio State University (USA)
genc.5@osu.edu

Michael D. Bond
Ohio State University (USA)
mikebond@cse.ohio-state.edu

Guoqing Harry Xu
UCLA (USA)
harryxu@cs.ucla.edu

Abstract

Byte-addressable persistent memory, such as Intel/Micron 3D XPoint, is an emerging technology that bridges the gap between volatile memory and persistent storage. Data in persistent memory survives crashes and restarts; however, it is challenging to ensure that this data is consistent after failures. Existing approaches incur significant performance costs to ensure crash consistency.

This paper introduces *Crafty*, a new approach for ensuring consistency and atomicity on persistent memory operations using *commodity hardware* with existing hardware transactional memory (HTM) capabilities, while incurring low overhead. Crafty employs a novel technique called *nondestructive undo logging* that leverages commodity HTM to control persist ordering. Our evaluation shows that Crafty outperforms state-of-the-art prior work under low contention, and performs competitively under high contention.

CCS Concepts: • **Information systems → Storage class memory**; • **Software and its engineering → Concurrent programming structures**.

Keywords: persistent transactions, transactional memory

1 Introduction

Non-volatile memory (NVM) technologies, such as phase change memory (PCM) [38, 56, 61], resistive random-access memory (RRAM) [55], spin-transfer torque memory (STT-MRAM) [35], and 3D XPoint [26], are designed to combine DRAM's byte-addressability and storage's durability: A program's updates to data structures residing in persistent memory can persist across failures such as a program crash or power interruption. As a result, NVM has the potential to fundamentally change the dichotomy between DRAM and durable storage in many important domains such as storage systems [24, 36, 58, 59, 63], databases [1, 2, 25, 64], and big data analytics [53].

State of the art. As with any storage system, the first challenge in effectively using NVM is to provide *crash consistency* [44, 52], which allows a program to correctly recover from persistent data upon a failure. Crash consistency is often achieved by leveraging *transactional support* in a high-level programming model. The developer specifies *persistent transactions*, in which updates to persistent memory appear to be one atomic unit—upon a program crash, either all or none of these updates are committed, ensuring that important data structures are always left in a consistent state.

However, prior work's mechanisms for persistent transactions have two main drawbacks. First, all of the mechanisms—undo logging [9, 32], redo logging [52], and copy-on-write [7, 15, 39, 43, 57]—incur performance costs such as persist latency on each write, lookups at program reads, maintenance of shadow memory, and poor multithreaded scalability.

Second, while commodity hardware transactional memory (HTM) such as Intel's transactional synchronization extensions (TSX) [21, 22, 60] is an appealing mechanism for supplementing persistent transactions to achieve full ACID transactions, persistent transaction mechanisms are *incompatible* with commodity HTM because of the following dilemma: *To ensure correct recovery, log entries must be persisted before a transaction commits, yet the nature of transactions dictates that executing transactions cannot perform persist operations.* Although some recent work shows how to make hardware transactions persistent [3, 4, 7, 15, 16, 39, 54], it has major drawbacks such as requiring log lookups at reads, using shadow memory, incurring scalability bottlenecks, or relying on nontrivial hardware changes (Section 2).

Contributions. This paper addresses *both* major limitations of prior work by *leveraging* commodity HTM to control persist ordering. We introduce a new kind of persistent transaction mechanism, *nondestructive undo logging*, that exploits commodity HTM to populate and persist undo logs before

59

making persistent writes visible. Key to nondestructive undo logging—which runs a persistent transaction's code in a hardware transaction and logs persistent writes in an undo log—is that the hardware transaction *rolls back* its persistent writes prior to committing, effectively creating its undo log entries without performing actual persistent writes. This behavior breaks HTM's persist–commit dependence cycle mentioned above, by decoupling the undo log updates from the persistent writes. After committing its hardware transaction that computes the undo log entries, a persistent transaction can perform its persistent writes—albeit in a way that is consistent with the persisted undo log entries and with other threads' persistent transactions.

We apply nondestructive undo logging in introducing *Crafty*, a novel and general approach for correct and efficient persistent transactions using *unmodified commodity HTM*. For each persistent transaction, Crafty first uses nondestructive undo logging to compute and persist undo log entries. It then performs the transaction's writes—by performing the logged writes directly if contention is low, or by repeating the transaction's execution while validating its consistency with the persisted undo log entry if contention is high. Crafty can operate in a *thread-unsafe* mode that provides only failure atomicity (relying on some other mechanisms such as locks for thread atomicity), or it can operate in a *thread-safe* mode that provides both failure and thread atomicity (i.e., full ACID transactions).

We implemented Crafty by extending the publicly available implementation of *NV-HTM* [7], which also implements *DudeTM* [39]; both approaches support persistent transactions with HTM using shadow-memory-based copy-on-write mechanisms. Our evaluation uses several programs with varying levels of thread contention: persistent transaction microbenchmarks and transactional benchmarks. Our results demonstrate that Crafty outperforms the two state-of-the-art HTM-based persistent transaction implementations NV-HTM and DudeTM, especially under low thread contention. Furthermore, Crafty usually adds low run-time overhead over *non-durable* transactions, and its overhead is largely thread local and thus scales well with additional threads.

These results suggest that nondestructive undo logging and Crafty are promising approaches for providing efficient, HTM-compatible persistent transactions.

2 Background and Motivation

This section covers background on persistent memory programming models and motivates the need for better mechanisms for persistent transactions.

2.1 Persistent Memory Programming Model

The key challenge of supporting persistent memory is ensuring that if a failure occurs, a *recovery observer* can restore persistent memory to a state that is usable by the restarted program. This property is often provided through *failure atomicity*: in the event of a crash or power failure, persistent memory state can be restored so that each *persistent transaction* appears to have executed fully or not executed at all [5, 7–9, 18, 23, 27, 39, 41].

In addition, a multithreaded program generally needs *thread atomicity*—persistent transactions execute atomically with respect to other threads—and state reconstructed by the recovery observer should be consistent with the commit order of persistent transactions. A program with persistent transactions can provide thread atomicity by using locks [8, 18, 23], or by using transactional memory [7, 9, 15, 39, 52]—in which case the transactions have full ACID properties.

Requirements. An implementation of persistent transactions must ensure that, after a crash, the recovery observer can restore the program's persistent state so that it corresponds to a serialization of persistent transactions consistent with the program's multithreaded execution. For example, if transaction *A happened before* transaction *B*, the recovered state must correspond to one of the following three execution scenarios: (1) *B* executed after *A*, (2) only *A* executed, or (3) no transaction executed at all.

Furthermore, recovered state should correspond to a point in time that is not too "far back" from the crash time. Otherwise, the amount of work that needs to be re-executed may be too large to be practical.

2.2 Persistent Transaction Mechanisms

Upon a crash or power failure, the recovery observer must reconstruct a state in which transactions appear to have executed fully or not at all. This challenge is exacerbated by the fact that stores do not reach persistent memory in their issuing order. This is because processor caches effectively buffer writes until eviction or explicit write-back of the dirty line to persistent memory.

To ensure that stores reach persistent memory in order, one can use *persist* operations. A persist operation consists of one or more flush operations that write back specified cache lines to persistent memory, followed by a drain operation that waits until the flush operations have completed. On x86, flush can be implemented with the CLWB (cache line write-back) instruction, and drain can be implemented with the SFENCE (store fence) instruction [49]. A persist operation is expensive because it incurs the roundtrip write latency of NVM, which is expected to be several hundreds of nanoseconds [38, 50]. Even if the NVM controller buffers persistent stores and includes the buffer as part of the persistence domain [49], persist latency (i.e., the time for roundtrip communication with the NVM controller) is still significant. If a commodity approach can be developed that amortizes persistent latency effectively across many persistent writes, one can make a case for removing the buffer from the persistence domain, simplifying future hardware designs.

(a) A persistent transaction	(b) Undo logging applied to (a)	(c) Redo logging applied to (a)	(d) Copy-on-write applied to (a)

```
failure_atomic {
  *p = 1;
   ... = *q;
  *r = 2;
}
```

```
undoLog.append(p, *p);
flush( last  log  entry );
drain;
*p = 1;
 ... = *q;
undoLog.append(r, *r);
flush( last  log  entry );
drain;
*r = 2;
undoLog.append(COMMITTED);
```

```
redoLog.put(p, 1);
 ... = redoLog.lookup(q);
redoLog.put(r, 2);
redoLog.append(COMMITTED);
foreach entry in redoLog
  flush(entry);
drain;
foreach <ptr, val> in redoLog
  *ptr = val;
```

```
*p = 1; // writes  shadow mem
redoLog.append(p, 1);
 ... = *q; // reads shadow mem
*r = 2; // writes  shadow mem
redoLog.append(r, 2);
redoLog.append(COMMITTED);
foreach entry in redoLog
  flush(entry);
drain;
foreach <ptr, val> in redoLog
  // Write to persistent addr
  *getPersistAddr (ptr) = val;
```

Figure 1. To providing failure atomicity for the persistent transaction in (a), a system uses one of the following crash-consistency mechanisms: (b) undo logging, (c) redo logging, or (d) copy-on-write. Initial values for all locations are 0.

Persistent transactions generally use one of the following three mechanisms to provide crash consistency: *undo logging* [9, 32], *redo logging* [52], and *copy-on-write* [7, 39, 43, 57]. Marathe et al. compared these mechanisms quantitatively and found that no mechanism is a clear winner in all situations (e.g., across thread counts or transaction sizes) [41]. Here we describe each mechanism and its drawbacks. We use Figure 1(a) as a simple example persistent transaction.

Undo logging. In undo logging, a persistent transaction logs the old value of a persistent location in a persistent undo log before the location is updated by a memory store. Undo logging enables fast read accesses: Since each store performs an *in-place* memory update, any memory load can directly read the latest value from persistent memory without being remapped to a different address. However, to ensure correct rollback after a crash, the implementation must persist (i.e., flush and drain) *each update to the undo log* before writing to the corresponding persistent memory location, incurring a high write latency for each NVM write.

Figure 1(b) shows how undo logging works for the persistent transaction in Figure 1(a). To signal the end of a persistent transaction's log entries, the implementation appends a COMMITTED entry to the undo log. A multithreaded implementation can include a timestamp (not shown in the figure) with the COMMITTED entry to enable the recovery observer to reconstruct a state corresponding to some globally consistent point in time.

Redo logging. In contrast, instead of performing in-place updates to persistent memory, redo logging *buffers* all persistent writes and performs them together at the end of the transaction. By buffering writes, redo logging pays the cost of persist ordering *once* only at the end of each transaction, effectively amortizing the latency across all of the writes. However, it adds an overhead for each persistent *read* because the read needs to find the latest value in a set of buffered writes. Since reads often significantly outnumber writes, redo logging can also incur significant overhead.

Figure 1(c) illustrates how redo logging works. Writes and reads to persistent memory are replaced with updates and lookups, respectively, to a map-based log so that reads of persistent memory correctly read from any preceding writes.

Copy-on-write. Recent work proposes *copy-on-write* mechanisms that maintain a *volatile shadow* for each persistent page to be modified [7, 39, 43, 57]. We focus on copy-on-write mechanisms that use shadow paging because it allows efficient in-place writes. Other copy-on-write mechanisms use indirection to copy an object upon the first write in a transaction, incurring costs similar to redo logging [15, 41]. Persistent transactions perform reads and writes normally, since virtual addresses are mapped to physical volatile shadow memory addresses. At the end of the transaction, changes to each shadow page are persisted to its corresponding non-volatile page. Figure 1(d) shows how this mechanism works.

Although copy-on-write techniques enjoy the performance benefits of undo and redo logging—and can be made compatible with commodity HTM as described shortly—shadowing the entire NVM is expensive and impractical. Most significantly, copy-on-write mechanisms must ensure consistency between the updates to volatile and non-volatile pages, leading to scalability bottlenecks, as detailed below.

2.3 Transactional Memory

A natural way to implement persistent transactions that provide full ACID properties is to leverage *transactional memory* [21, 22]. Much of the existing work on persistent transactions extends *software* transactional memory (STM) [20], which incurs a high overhead in detecting and resolving conflicts between concurrent transactions.

Hardware transactional memory (HTM), which detects and resolves conflicts at the hardware level, is an appealing technique for implementing efficient persistent transactions.

However, *commodity HTM* implementations including Intel's *restricted TM* (RTM) [60] are fundamentally incompatible with persistency. Because log updates must occur before memory updates, there is an obvious dilemma: On the one hand, undo or redo log entries must be persisted before the hardware transaction commits the actual memory updates (to ensure crash consistency), while on the other hand, the nature of the transaction dictates that these log entries cannot be persisted before the transaction commits—otherwise they cannot be revoked upon an abort. The updates to the log entries and the actual memory updates depend on each others, forming a dependence cycle that seemingly thwarts the use of HTM for persistent transactions.

Recent approaches use commodity HTM for persistent transactions, by *decoupling* persistence from HTM's concurrency control. *DudeTM* [39] and *NV-HTM* [7] show how copy-on-write mechanisms can support HTM-based persistent transactions. Hardware transactions perform *in-place* reads and writes to shadow memory. After a transaction commits, redo log entries can be persisted before copying the transaction's writes to persistent memory. In addition, by writing redo log entries and program writes to persistent memory asynchronously, writes to the same persistent locations can be combined.

The drawback, though, is that DudeTM and NV-HTM have significant disadvantages in maintaining shadow state and keeping updates to persistent memory consistent with the order of transactions writing to the volatile shadow state. First, these approaches add space overhead by maintaining two copies of program state, as discussed above. Second, they must ensure consistency between the order of the writes to DRAM inside a transaction and that to NVM at the end of the transaction. DudeTM computes timestamps by incrementing a global variable in commodity HTM, making it effectively incompatible with commodity HTM [39].

NV-HTM, on the other hand, works with unmodified commodity HTM, but it has two major scalability bottlenecks that limit performance at higher thread counts [7]. First, each persistent transaction cannot complete until *every other ongoing transaction* completes. In particular, each transaction cannot write a COMMIT entry to its redo log until it ensures that no ongoing transaction may still write a COMMIT entry for an earlier transaction, since redo logs are used by the recovery observer to roll the persistent state forward after a crash. Waiting ensures that if the recovery observer sees a COMMIT entry for a transaction, it sees COMMIT entries for all earlier transactions. Of course, this incurs overhead.

Second, threads that persist logs and program writes to persistent memory must do it in a serialized manner. In NV-HTM, an asynchronous background thread applies transactions' writes (based on their redo log entries) to persistent memory locations in timestamp order. This serialization of

writes to persistent memory is inherent in the fact that transactions record a timestamp (for efficiency), from which only a global transaction order can be inferred.

The DudeTM paper surmises that decoupling persistence from HTM may be "the best (and possibly the only) way to avoid the drawbacks of both undo and redo logging and reduce the performance penalty" [39]. Our work seeks to *counter* that supposition and overcome the performance disadvantages of existing persistent transaction mechanisms.

3 Crafty Overview

As Section 2.3 explained, the main obstacle that precludes efficient use of commodity HTM in implementing persistent transactions is the dependence cycle that results from the tight coupling of log entry updates and program memory updates: If a hardware transaction contains a mix of these two types of updates, it can neither commit before persisting, nor persist before committing.

To address this problem, we introduce a new persist transaction design called *Crafty* that leverages a new logging mechanism called *nondestructive undo logging*. Key to Crafty's success is breaking the persist–commit dependence cycle by executing the log entry updates and the program memory updates in *separate hardware transactions*, effectively decoupling these two types of updates. In nondestructive undo logging, a hardware transaction performs a LOG phase that executes a persistent transaction in a way that updates only undo log entries, not the program's persistent data. These log entries are persisted after the hardware transaction commits. Next, Crafty executes the program writes using another hardware transaction. These writes are performed in a way that is consistent with the updates of the log entries and also with other threads' executed transactions.

Challenges and insights. Achieving a correct and efficient design presents three major challenges. The *first challenge* is how to make the LOG phase only update undo or redo log entries without modifying program memory locations. To overcome this challenge, Crafty uses undo logging when executing the LOG phase: Before each write to a persistent memory location, the old value in the location is recorded in a *thread-local undo log*. At the end of the transaction, Crafty *rolls back* all of these writes by applying the entries of the undo log in a *reverse order*, effectively setting the modified values back to the their original values before the transaction executed. Furthermore, during this rollback process, when both the old and new values are visible, the hardware transaction builds a *redo log* for these locations. After the LOG phase commits, all of the undo log entries are persisted into persistent memory. Figure 2 shows how the LOG phase uses nondestructive undo logging to construct an undo log for the persistent transaction from Figure 1.

The *second challenge* is how to execute the program's memory updates in the same order as the updates to log

```
HTM_BEGIN;
undoLog.append(p, *p);
*p = 1;
 ...  = *q;
undoLog.append(r, *r);
*r = 2;
foreach entry <ptr, oldVal> in undoLog in reverse
  redoLog.append(ptr, *ptr);
  *ptr = oldVal; // undo each write
HTM_END;
foreach entry in undoLog
  flush(entry);
drain; // persist the undo log entries
/* ... Transaction's writes can now be performed here ... */
undoLog.append(COMMITTED);
```

Figure 2. How Crafty's crash-consistency mechanism, non-destructive undo logging, provides failure atomicity for the persistent transaction in Figure 1(a).

entries. To do this, Crafty starts the second phase, which updates program memory locations. In theory, all we need is a REDO phase that applies the redo log constructed at the end of the LOG phase. This naïve approach would work if persistent transactions were protected by a pessimistic technique such as locking, because transactions executed by different threads would conflict with each other. However, if persistent transactions can conflict, then a thread's LOG and REDO phases—which executed in two separate hardware transactions—may not execute together *atomically*. This can potentially lead to inconsistencies between the log entries and the contents in their corresponding memory locations. To solve this problem, Crafty lets the REDO phase check a conservative *conflict constraint* based on timestamps. Failure of this check is a *necessary but insufficient* condition for a transaction conflict. To guarantee safety, Crafty aborts the HTM transaction that executes this REDO phase.

The *third challenge* is what to do if and when REDO aborts. Due to the conservative nature of our conflict constraint, a REDO abort does not necessarily indicate a real conflict. Hence, if and only if REDO aborts, Crafty executes a *VALIDATE* phase, which *re-executes the persistent transaction* to check the validity of the undo log entries that were persisted in the LOG phase. If all of the undo log entries are still valid, the transaction succeeds, allowing the memory writes to be committed and visible to other threads. Any mismatch between the values in a log entry and its corresponding memory location makes VALIDATE abort, indicating that another thread has committed new, conflicting writes after the current thread's LOG phase finished. The aborted thread handles this case by starting over—by re-executing the LOG phase and constructing new undo and redo logs.

Outline. Section 4 describes how Crafty executes persistent transactions to provide atomicity at run time and support recovery on a crash. Section 5 describes how recovery restores persistent state after a crash.

4 How Crafty Executes Transactions

This section describes how Crafty leverages nondestructive undo logging to execute persistent transactions.

Execution modes. Crafty can operate in either of two modes. In its *thread-safe mode* (this paper's focus), programmers specify persistent transaction boundaries, and Crafty provides both atomicity and durability (i.e., all ACID properties) for persistent transactions.

Crafty's *thread-unsafe mode* is appropriate when locks or another mechanism already provides atomicity, so Crafty only needs to provide durability. In this mode, programmers can specify transaction boundaries explicitly or inform Crafty to treat all critical sections [5, 8, 23, 27, 40] or synchronization-free regions [18] as persistent transactions.

Figures 3 and 4 show how Crafty operates in its thread-safe and thread-unsafe modes (Section 3), respectively. Sections 4.1–4.3 provide a detailed description of the LOG, REDO, and VALIDATE phases in the context of Crafty's thread-safe mode. In thread-safe mode, repeated aborts cause Crafty to transition to thread-unsafe mode while holding a *single global lock* (SGL), as Figure 3 shows and Section 4.4 describes.

The rest of this section uses Figure 5 as an example to show how Crafty works.

4.1 LOG Phase

Crafty's LOG phase generates undo log entries for an executed persistent transaction and then persists these entries. The key treatment here is that the LOG phase does *not* commit any program writes to persistent memory. The LOG phase achieves this outcome by allowing the persistent transaction to perform writes normally during its execution, but rolling back the writes before the hardware transaction commits.

Algorithm 1 shows the details of the LOG phase, which executes the persistent transaction body in a hardware transaction. Before each persistent write, the LOG phase records the old value of the written-to address in the executing thread's persistent undo log. For example, in Figure 5, each persistent transaction's LOG phase adds old values to the undo log before each write.

At the transaction end, the LOG phase uses the undo log entries to roll back the transaction's writes, by applying the undo log entries' old values in the reverse order. When rolling back the writes, Crafty simultaneously builds a volatile *redo log* for the transaction, which can be used by the subsequent REDO phase to perform program writes. For example, in Figure 5, starting from the "Start roll back:" comment, the persistent transaction's LOG phase rolls back the writes by applying the values from the undo log. Before committing the

Start persistent transaction
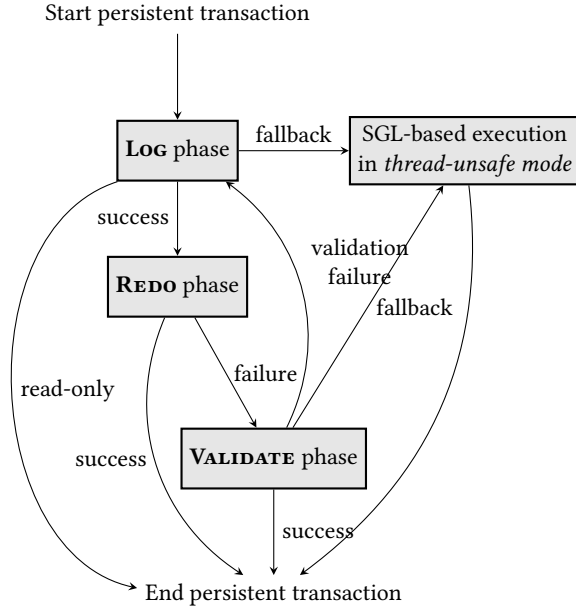
Figure 3. Crafty's phases in *thread-safe* mode.

Start persistent transaction;
$k \leftarrow$ MAX_WRITES
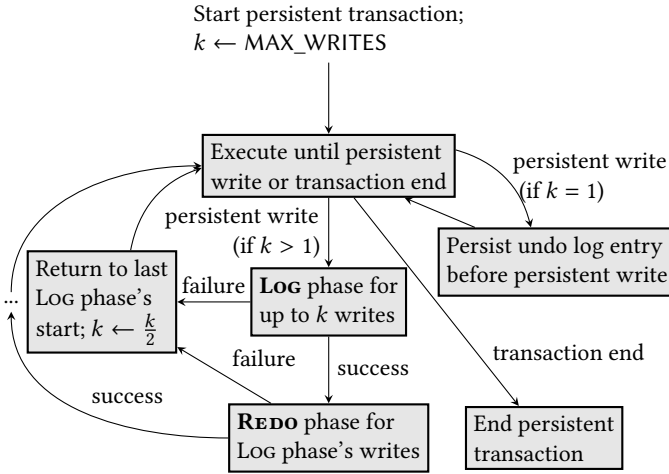
Figure 4. Crafty's phases in *thread-unsafe* mode.

hardware transaction, the LOG phase adds a LOGGED entry with a Lamport timestamp[1] denoting the current logical time (which is equivalent to the logical time at the beginning of the hardware transaction since HTM ensures atomicity). The timestamps will be used by recovery to order undo log entries by different threads. In Figure 5, each LOG phase concludes by inserting a LOGGED entry into the undo log before committing the hardware transaction.

After committing the transaction, the LOG phase flushes the transaction's undo log entries to persistent memory. The algorithm flushes the transaction's undo log entries but does

---

[1]If two events are ordered by happens-before, their logical times are correspondingly ordered [37].

Thread 1	Thread 2
atomic_and_durable {	**atomic_and_durable** {
*p = *q;	*q = 2;
*r = 1;	*s = 3;
}	}

(a) Two persistent transactions.

Thread 1

LOG phase:
HTM_BEGIN
undoLog$_{T1}$.add(p, 0)
*p = *q
undoLog$_{T1}$.add(r, 0)
*r = 1
// Start roll back:
redoLog$_{T1}$.add(r, 1)
*r = 0 // from undo log
redoLog$_{T1}$.add(p, 0)
*p = 0 // from undo log
lastTS$_{T1}$ = ts ()
undoLog$_{T1}$.add(LOGGED,
           lastTS$_{T1}$)
HTM_END

REDO phase:
HTM_BEGIN
check gLastRedoTS < lastTS$_{T1}$
*p = 0 // from redo log
*r = 1 // from redo log
gLastRedoTS = ts ()
undoLog$_{T1}$.add(COMMITTED,
         gLastRedoTS)
HTM_END

Thread 2

LOG phase:
HTM_BEGIN
undoLog$_{T2}$.add(q, 0)
*q = 2
undoLog$_{T2}$.add(s, 0)
*s = 3
// Start roll back:
redoLog$_{T2}$.add(s, 3)
*s = 0 // from undo log
redoLog$_{T2}$.add(q, 2)
*q = 0 // from undo log
lastTS$_{T2}$ = ts ()
undoLog$_{T2}$.add(LOGGED,
           lastTS$_{T2}$)
HTM_END

REDO phase:
HTM_BEGIN // redo
check gLastRedoTS < lastTS$_{T2}$
ABORT // check failed

VALIDATE phase:
HTM_BEGIN
check *q == 0 // from undo log
*q = 2
check *s == 0 // from undo log
*s = 3
check # writes == # log entries
gLastRedoTS = ts ()
undoLog$_{T2}$.add(COMMITTED,
         gLastRedoTS)
HTM_END

(b) A possible execution of the persistent transactions in (a) using Crafty in its thread-safe mode, which provides all ACID properties.

Figure 5. An example of Crafty's thread-safe mode executing persistent transactions. Initial values of *p, *q, *r, and *s are 0. The example omits flush and drain instructions.

Algorithm 1   Log phase

1: HTM_BEGIN
    ▷ Start of persistent transaction
    …
    ▷ Program write to persistent variable:
2: Add ⟨addr, oldValue⟩ to T.undoLog   ▷ T is current thread
3: *addr = newValue   ▷ Original program write
    …
    ▷ End of persistent transaction
4: Roll back transaction's writes using T.undoLog, and populate local redo log from T.undoLog
5: Add ⟨LOGGED, getTimestamp()⟩ to T.undoLog
6: HTM_END
7: flush(T.undoLog entries for this transaction)

Algorithm 2   Redo phase

    Thread-safe Redo phase:
1: HTM_BEGIN
2: **if** gLastRedoTS   <   LOGGED timestamp from Log phase **then**
3:     gLastRedoTS ← getTimestamp()
4:     Perform thread-unsafe Redo phase
5: **else**
6:     Abort transaction and fail Redo phase
7: **end if**
8: HTM_END
9: flush(written-to addresses)

    Thread-unsafe Redo phase:
10: Perform writes from redo log
11: **if** not in hardware transaction **then**
12:     flush(written-to addresses)
13: **end if**
14: Add ⟨COMMITTED, getTimestamp()⟩ to T.undoLog

---

not wait for them to be written back to persistent memory (i.e., flush but no drain) because the program writes will be committed by the Redo or Validate phase inside of a hardware transaction, which has drain semantics (e.g., an RTM transaction has SFENCE semantics).

Once the Log phase completes, undo log entries for the transaction have been persisted, but the transaction has effectively *not* executed from the perspective of other threads and memory because none of the memory updates have been performed. In Figure 5, after the Log phase completes, *p, *r, *q, and *s still have their initial values (0). To make these updates visible to other threads and persistent memory, Crafty uses the Redo or, if needed, the Validate phase.

A read-only transaction need not add a LOGGED entry to the undo log or perform any persist operations, and it can skip the Redo and Validate phases, as shown in Figure 3.

4.2 Redo Phase

The Redo phase applies the writes from the redo log (in the reverse order of how they were recorded in the Log phase), as illustrated in Algorithm 2.

If the program is single-threaded or no other threads access persistent memory, it is safe to execute the Redo phase *unconditionally.* However, if multiple threads are executing persistent transactions, atomicity can be violated. For example, thread B's Redo phase can occur between thread A's Log and Redo phases. Hence, it is important to ensure that *A's Redo phase completes only if it executes atomically with its preceding Log phase.*

To this end, Crafty uses a global variable gLastRedoTS that represents the timestamp of the last writes committed by any thread. Figure 5 demonstrates how gLastRedoTS is updated and used. Crafty checks gLastRedoTS at the start of the Redo phase in Thread 1. The check succeeds because no thread has committed writes since Thread 1's Log phase. The Redo phase then performs the writes from the redo log. Thread 1 completes the Redo phase by updating gLastRedoTS and adding a timestamped COMMITTED entry to the log. This

timestamp represents the time at which the transaction's writes happened in relation to other threads' transactions.

A failed check indicates a potential atomicity violation. In Figure 5, Thread 2's check of gLastRedoTS fails because Thread 1 updated gLastRedoTS to reflect that it committed writes (in its Redo phase) *after* Thread 2's Log phase. Thread 2's Redo phase thus fails, and Crafty tries the Validate phase. (Alternatively, under different timing, Thread 2's Redo phase could start and complete before Thread 1's Redo phase read gLastRedoTS, allowing Thread 2 to commit its writes with the Redo phase. Thread 1 would fail the Redo phase and try the Validate phase, re-executing the transaction and writing an updated value of *q, 2, to *p.)

If successful, the Redo phase concludes by flushing the transaction's writes to persistent memory, but does not wait for the write-backs to finish (i.e., flush but no drain). The next persistent transaction's Log phase will perform a hardware transaction, which has drain semantics, and the recovery algorithm always rolls back each thread's latest transaction in case its writes had not fully persisted (Section 5).

The Redo phases of all transactions are effectively serialized. This does not necessarily cause a bottleneck in performance because the Redo phase is often short and can execute concurrently with Log and Validate phases.

4.3 Validate Phase

The goal of the Validate phase is to execute a persistent transaction that is consistent with the persisted undo log entries. The Validate phase checks consistency by comparing the old values recorded in the undo log with the current values of the same locations, as Algorithm 3 illustrates.

The Validate phase checks whether, for each program write, its corresponding entry in the undo log matches the write's address and the value at the address. If it does, this

---

Algorithm 3   VALIDATE phase

1: Reset T.undoLog to the beginning of the transaction
2: HTM_BEGIN
   ▷ Start of the persistent transaction
   . . .
   ▷ Program write to a persistent variable:
3: **let** ⟨expectedAddr, expectedValue⟩ be next entry in T.undoLog
4: **if** addr ≠ expectedAddr ∨ *addr ≠ expectedValue **then**
5:     Abort transaction and fail validation
6: **end if**
7: *addr ← newValue   ▷ Original program write
   . . .
   ▷ End of the persistent transaction
8: Check that the next entry in T.undoLog is a LOGGED entry (if
   not, abort the transaction and fail the validation)
9: gLastRedoTS ← getTimestamp()
10: Add ⟨COMMITTED, gLastRedoTS⟩ to T.undoLog after the
    LOGGED entry
11: HTM_END
12: flush(written-to addresses)

---

implies the validity of the undo log entry. For example, in Figure 5, Thread 2's VALIDATE phase checks that both writes to q and s match the original addresses and old values in the undo log, and that there are no new writes. At the end of the persistent transaction, the hardware transaction is committed, and the writes are persisted and made visible to other threads. Note that it is important to re-execute the transaction—by validating the undo log entries rather than just performing the writes—to ensure (implicitly) that values *read* by the transaction are still consistent with the undo log entries. Like the REDO phase, after performing the writes, Crafty adds a ⟨COMMITTED, getTimestamp()⟩ entry to the log, which represents the time at which the transaction's writes happened in relation to other threads.

Note that the VALIDATE phase executes only if the REDO phase fails, Every persistent transaction commits its writes exactly once, with either the REDO or VALIDATE phase.

4.4 Single Global Lock Fallback

A hardware transaction may abort for a variety of reasons including a conflict with other threads, cache capacity overflow, or an unsupported event such as an interrupt [60]. Since commodity HTM generally provides no progress guarantee, special care must be taken to ensure that an execution makes progress. As Figure 3 shows, Crafty's thread-safe mode retries an aborted transaction several times; if no REDO or VALIDATE phase commits successfully, it falls back to acquiring a *single global lock (SGL)* to provide progress guarantees. The SGL serves two purposes. First, it eliminates conflicts among different threads. Second, it allows Crafty to execute in *thread-unsafe mode*, where Crafty can execute shorter hardware transactions (fewer instructions) or without hardware transactions to ensure progress.

The SGL is a global variable that a thread acquires by updating it atomically from 0 to 1, and releases by setting it to 0 (with proper memory fencing). To ensure consistency with respect to other threads executing hardware transactions in thread-safe mode, each hardware transaction in thread-safe mode must check whether the SGL is 0 at the beginning of the transaction; if the SGL is 1, the transaction must abort (not shown in the algorithms). This handling ensures consistency with an ongoing SGL section or with an SGL section that starts while the transaction is executing (since the transaction's read set contains the SGL). This fallback method is referred to as *speculative lock elision* in the literature and has been widely studied [28, 46, 60].

After acquiring the SGL, a thread can safely execute in Crafty's thread-unsafe mode, as illustrated in Figure 4. In this mode, the SGL ensures atomicity, so HTM serves solely to implement nondestructive undo logging (i.e., to prevent updated cache lines from being written back to persistent memory prematurely), *not* for thread atomicity.

As a result, thread-unsafe mode uses hardware transactions for the LOG phase only, which can wait to start a hardware transaction until the first persistent write of the persistent transaction. Thread-unsafe mode does not use HTM for the REDO phase because no other threads can update the global timestamp gLastRedoTS, and hence this phase always succeeds. The VALIDATE phase is not needed at all.

Ensuring progress. Even without contention from other threads, a hardware transaction may still abort for cache capacity or other reasons. The LOG phase in thread-unsafe mode can ensure progress by breaking a persistent transaction into smaller hardware transactions, each executing at most k persistent writes. After executing k persistent writes (or fewer, if the persistent transaction ends before k is reached), the LOG phase completes normally, rolling back writes and persisting undo log entries including a LOGGED entry. The REDO phase then performs the k (or fewer) persistent writes, except that it does *not* add a COMMITTED entry, which should only be used to indicate the end of the (SGL-based) transaction. The LOG and REDO phases continue executing the persistent transaction, in chunks of up to k writes. If $k = 1$, the LOG phase writes and persists an undo log entry *before* performing the write to memory, without using any hardware transaction.

When entering thread-unsafe mode for a persistent transaction, Crafty begins with a (relatively large) value of k (e.g., 64) with the goal to amortize persist latency across multiple writes. After each transactional abort in thread-unsafe mode, Crafty decreases k geometrically for the next hardware transaction. When the value of k drops to 1, thread-unsafe mode is guaranteed to make progress. Figure 4 illustrates the logic for thread-unsafe mode, which executes the transaction in k-write chunks until completion.

Before releasing the SGL, Crafty adds a COMMITTED entry to the persistent undo log. All of the SGL section's hardware transaction's LOGGED and COMMITTED entries use the *same* timestamp (from the first call to getTimestamp()) to ensure that the recovery algorithm either rolls back all or none of the SGL section's writes.

Crafty thus adaptively adjusts transaction sizes to provide a tradeoff between persist latency and the risk of aborting. Prior work in other contexts splits transactions to balance between per-transaction costs and aborts costs [42, 51].

5 How Crafty Recovers After a Crash

This section presents Crafty's recovery logic. We first describe how recovery is done under the assumptions that *an infinite log is initially zeroed (i.e., no wraparound or reuse)* and *log entries are persisted atomically*. Then we show how to handle logs without these simplifying assumptions.

5.1 Basic Recovery Logic

Under the above-mentioned assumptions, the recovery observer can detect *persisted entries*, which are entries with a nonzero addr field (which is either an address or a LOGGED or COMMITTED tag). A *fully persisted sequence* is a consecutive sequence of persisted ⟨addr, oldValue⟩ entries preceded by a (persisted) LOGGED or COMMITTED entry and concluded by a persisted LOGGED entry.

The recovery observer needs to roll back the last *fully persisted sequence of each thread* because some of the corresponding writes may have persisted, but not all of them have definitely persisted. Let ⟨LOGGED, ts⟩ be the sequence's concluding entry. Then we define a sequence's timestamp to be ts. To arrive at a globally consistent snapshot, the recovery observer must also roll back every sequence that has a timestamp *later than or equal to* the timestamp of any sequence being rolled back. Any persisted entries outside a fully persisted sequence must not be rolled back because their corresponding writes definitely have not persisted.

To roll back a sequence, the recovery observer applies the ⟨addr, oldValue⟩ entries in reverse order, performing *addr = oldValue for each entry. The recovery observer rolls back the fully persisted sequences in the reverse timestamp order.

5.2 Handling High-Performance Logs

Next, we discuss how Crafty provides correct recovery in the absence of simplifying assumptions about the logs. The following design handles circular logs that reuse entries, and it does not require a log entry to be persisted atomically. The design also addresses a limitation of Crafty's design as presented so far: Because the recovery observer rolls back at least each thread's last transaction, a rolled-back transaction can be arbitrarily far back in time if a thread has not executed a transaction in a while.

The design assumes that the system provides persistence at *word* (or coarser) granularity. The design relies on each thread's circular log being large enough to hold log entries for at least two persisted sequences (which are bounded due to HTM bounding constraints).

Distinguishing reused entries. To handle reuse of undo log entries (e.g., via a circular log), the recovery observer needs to be able to tell whether an entry ⟨addr, oldValue⟩ is from the latest transaction or the last wraparound of the log. Inspired by prior mechanisms [10], Crafty's execution of transactions maintains a per-thread *wraparound bit* that is encoded in each word and flips each time the log wraps around. This wraparound bit allows recovery to differentiate words written *after* versus *before* the latest log wraparound. Because logged transactions occur more often than wraparound, recovery can only observe log entries that are *after* the next-to-latest wraparound and hence a single wraparound bit suffices.

We further assume that all addresses are word (4- or 8-byte) aligned. This allows us to steal two or three bits of the addr word in each ⟨addr, value⟩ log entry. One of them is used as the wraparound bit. The LOGGED and COMMITTED tags are each represented as a reserved, aligned address.

Because NVM is only guaranteed to provide persistence at word granularity, the value word in a ⟨addr, value⟩ log entry will also need a wraparound bit. However, value needs all of its bits for program values. We thus steal another bit in each addr word to store a bit (e.g., the lowest bit) of the value word, allowing that same bit of the value word to be replaced with the wraparound bit.

Discarding entries and bounding rollback severity. In order for Crafty's Log phase to reuse log entries (e.g., via a circular log), we must be able to discard some entries that would no longer be needed by the recovery observer. Since Crafty must not discard entries for a logged transaction that needs to be rolled back, we need to ensure that the earliest possible rollback timestamp ts is greater than the timestamp of a logged transaction that Crafty is ready to discard. A related issue we address here is *bounding* how far back in time the recovery observer must roll back to. This distance can be quite far if a thread has not executed a persistent transaction for a while.

The logging algorithm maintains a global timestamp tsLowerBound that is a lower bound on the earliest possible timestamp r that recovery might need to roll back to. It is a lower bound because, for performance reasons, it is kept up to date *lazily*. When adding a LOGGED entry to its undo log, a thread checks that

$$currentTS() < tsLowerBound + MAX\_LAG$$

where MAX_LAG represents a customizable maximum time duration for which recovery might need to roll back, and currentTS() is a timestamp representing the current time.

Likewise, whenever a thread T gets halfway through its circular log, it first checks if overwriting the next half of the log will violate

$$T.log.earliestTSToBeOverwritten > tsLowerBound$$

If either of these conditions fails, thread T performs further inspection, by checking the following two conditions for every other thread U:

$$currentTS() < U.lastCommittedTxn.ts + MAX\_LAG$$

$$T.log.earliestTSToBeOverwritten < U.lastCommittedTxn.ts$$

T can perform these checks safely (atomically) by executing them in a hardware transaction, performing read-only accesses to U.logStart and U's log. If either condition fails on U, then T forces U to append an by committing a ⟨LOGGED, getTimestamp()⟩ entry to U's log (representing an empty completed transaction). T can accomplish this by using a transaction to safely update U's log (because we need to be careful about interference with U, especially its non-transactional state manipulations).

After T makes each delinquent thread U commit a more up-to-date transaction, it sets

$$tsLowerBound \leftarrow \min_U U.lastCommittedTxn.ts$$

Note that most transactions only need to read a global shared variable (tsLowerBound) that is mostly read-only, resorting to more expensive operations only when they are halfway through the circular undo log. The frequency of expensive operations can be reduced by increasing the size of each thread's circular log.

Providing immediate persistence. Some persistent transaction systems, including DudeTM [39] and NV-HTM [7], guarantee that if a persistent transaction completes and the thread continues execution, then the recovered state will include the completed transaction's state. This "immediate persistence" property ensures that the persistent state is consistent with any externally visible, irrevocable actions between transactions such as system calls. However, Crafty does *not* provide "immediate persistence" because it does not ensure that all writes have been persisted when a transaction completes (which is why recovery rolls back each thread's last logged transaction). Some prior work including PMThreads [57] also does not provide immediate persistence.

Instead of providing immediate persistence, Crafty can provide a method for "on-demand" immediate persistence (to be invoked before performing externally visible, irrevocable actions). Crafty can implement on-demand immediate persistence by adding an ⟨LOGGED, getTimestamp()⟩ entry to each thread's, similar to the approach described above for reusing log entries and bounding rollback severity. Modifying other threads' logs can be performed safely by executing in a hardware transaction. Our prototype implementation does not suppport on-demand immediate persistence.

6 Implementation

Our Crafty implementation, which we have made publicly available,[2] extends the publicly available NV-HTM implementation [7].[3] The NV-HTM implementation also provides a configuration that represents the prior work DudeTM [39]. It also includes an *Non-durable* configuration that simply executes each persistent transaction in a hardware transaction and thus does not provide any crash-consistency guarantees.

The NV-HTM implementation *emulates* non-volatile memory in volatile memory by performing 300 ns of busy waiting at drain operations (emulating the roundtrip latency of each SFENCE instruction that follows one or more CLWB instrutions). This methodology is consistent with the evaluations of prior work including DudeTM and NV-HTM [7, 39].

Crafty logging details. Each thread has an undo and a redo log. Undo logs are in non-volatile memory and are circular. Redo logs are in volatile memory and not needed after a persistent transaction completes, so the next persistent transaction can reuse the redo log from the beginning.

Each undo log entry ⟨addr, oldValue⟩ contains two 8-byte words: the written-to address and the old value. Each addr value is 8-byte-aligned because all writes are expressed as 8-byte, aligned stores. The implementation merges the LOGGED and COMMITTED entries into a single entry, overwriting the entry's timestamp on commit. This optimization is safe as the recovery observer does not need to differentiate between LOGGED and COMMITTED entries when deciding what sequences to roll back. The recovery observer can check if each log entry has persisted using the wraparound bit. Timestamps come from RDTSC.

The implementation performs the work needed to support rollback (i.e., the wraparound bit and the log checks in Section 5.2). However, we have *not* implemented the actual recovery logic, leaving it and its evaluation to future work.

Mixed-mode accesses. The implementation requires that all writes to persistent memory happen in persistent transactions. Crafty can support writes to volatile memory in transactions by ensuring transactions are idempotent with respect to volatile memory accesses. Our implementation requires manual transformation of transactions to make them idempotent with respect to *function-local* variables. It does not allow other volatile memory writes in transactions, but could do so by adding undo logs for volatile accesses.

The same (volatile or persistent) variable can be accessed both in and out of transactions, subject to the aforementioned constraints. Programmers must be careful to synchronize such accesses correctly: Although Intel's RTM provides strong atomicity [21], Crafty may fall back to using a global lock for providing thread atomicity. Programs thus must ensure transactional data race freedom [12].

---

[2]https://github.com/PLaSSticity/Crafty
[3]https://bitbucket.org/dfscastro/nvhtm-selfcontained/src/master/

Memory management. Because the LOG and VALIDATE phases execute the same code, the implementation must handle side effects from malloc and free to avoid leaking memory and failing checks in the VALIDATE phase. The implementation thus logs allocations during the LOG phase and reuses the allocated memory at corresponding malloc calls during the VALIDATE phase. Similarly, the LOG phase logs free calls during the LOG phase, and either performs the logged frees after completing the REDO phase or allows the VALIDATE phase to performed free calls and then discards logged frees.

7 Evaluation

This section evaluates the performance of Crafty by comparing it with prior work's HTM-compatible persistent transactions and with non-durable transactions.

7.1 Methodology

Configurations. Our experiments run Crafty in thread-safe mode to provide full ACID transactions. NV-HTM and Dude-TM are run under their standard configurations. As a baseline, we run the implementation's *Non-durable* configuration that does not provide any guarantees on crash consistency.

In addition to evaluating Crafty's full-blown version (in thread-safe mode), we evaluate two variants of Crafty that exclude the VALIDATE and REDO phases, referred to as *Crafty-NoRedo* and *Crafty-NoValidate*, respectively, in the rest of the section. These configurations help tease out the performance effects of Crafty's components. Note that these configurations are also fully functioning configurations that provide the same guarantees as Crafty.

Evaluated programs. We use two microbenchmarks and a set of transactional memory benchmarks.

The *bank* microbenchmark is from the publicly available NV-HTM implementation [7] that performs random transfers between accounts. We configure the benchmark to run five transfers (ten persistent writes) per transaction with three levels of contention: *high*, *medium*, and *no conflict*. The difference in conflict rates is achieved by varying the number of accounts—the medium- and high-conflict configurations operate on 4,096 and 1,024 cache-line-aligned accounts, respectively. The no-conflict configuration avoids all conflicts by partitioning the accounts among threads.

The other microbenchmark performs operations on a B+ tree and is adapted from the implementation of Zardoshti et al. [62] by annotating writes to shared memory. The benchmark provides two variants: one performs only insertions on the tree, and the other performs a mixture of lookups, insertions, and removals.

As a standard benchmark suite for transaction memory research, our experiments use the transactional *STAMP benchmarks* [6]. In particular, we consider each transaction to be a persistent transaction, and treat all shared-memory accesses in transactions as accesses to persistent memory. This same

methodology was used in the evaluation of prior work [7]. We exclude the benchmark yada as it fails to run with Non-durable and NV-HTM due to a pointer corruption, and bayes because around half of the transactions fall back to the SGL mode due to HTM incompatible instructions which makes the results not meaningful.

Experimental setup. We run the experiments on a machine with a quiet 16-core Intel Skylake processor with hyper-threading disabled. The implementation uses native hardware transactions [48] and emulates non-volatile memory as described in Section 6. Each reported result is the arithmetic mean of five trials. The throughput results are normalized to the throughput of the single-thread, non-durable configuration of the same benchmark. We define throughput as the inverse of the execution's wall-clock time.

7.2 Performance Results

This subsection presents our main results: performance and scalability for the evaluated programs across thread counts and persistent transaction implementations. We also perform additional measurements that help explain the performance including (1) breakdowns of persistent transactions by Crafty phases and (2) hardware transaction commit and abort counts and abort causes. These additional measurements, as well as performance results that emulate 100 ns (instead of 300 ns) write latency, can be found in an extended arXiv version [14].

Bank microbenchmark. Figure 6 compares Crafty and other persistent transaction implementations, under different contention levels. The general trend behind these results is that Crafty outperforms NV-HTM and DudeTM under low-contention settings, when there are few threads or few conflicting transactions. For example, under all contention levels Crafty outperforms NV-HTM and DudeTM for 1–2 threads.

Under high contention, Crafty scales poorly because it Crafty amplifies the transactional conflicts by executing persistent transactions using more hardware transactions than other approaches. While NV-HTM scales well up to 4 threads, its scalability limitations (Section 2) cause it to anti-scales above 4 threads and underperforms Crafty above 8 threads. Crafty outperforms or performs as well as the competing approaches except for NV-HTM on the high-conflict configuration at 4 threads.

Note that NV-HTM's and DudeTM's throughput drops dramatically at 16 threads. NV-HTM and DudeTM use an extra thread that performs writes to persistent memory. Because 16 program threads are running on 16 cores, the extra thread is scheduled on the same core as a program thread, causing frequent context switches because of the producer–consumer relationship between the two threads.

Figure 6(c) motivates the VALIDATE phase. When the number of threads is above 4, Crafty-NoValidate is slower than Crafty because REDO fails due to timestamp checks, but
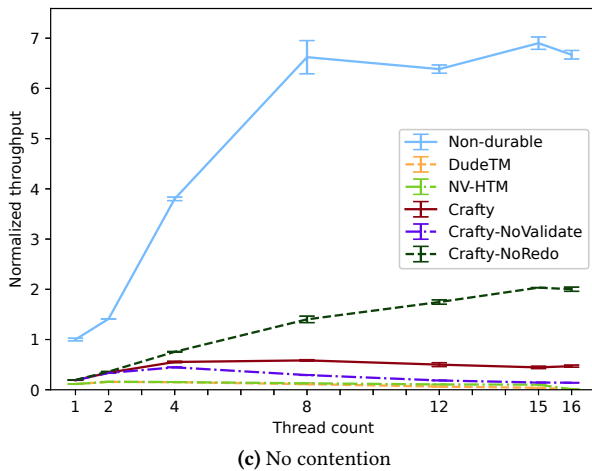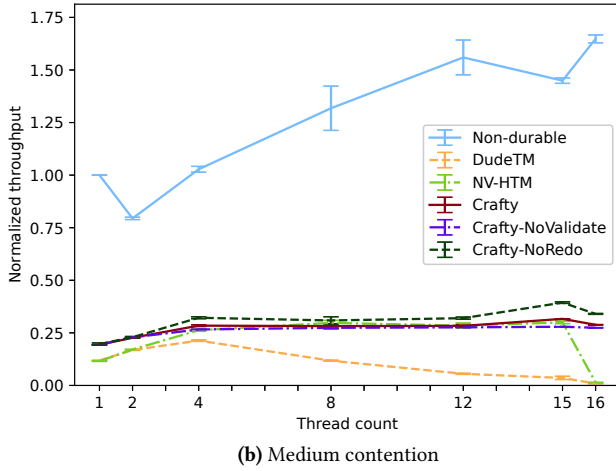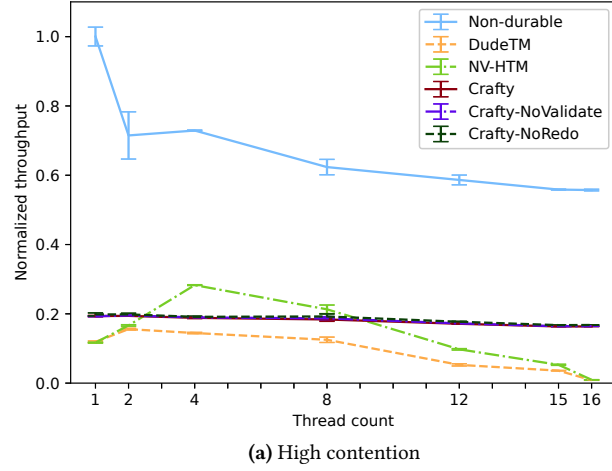
(a) High contention

(b) Medium contention

(c) No contention

Figure 6. Throughput of Crafty and competing approaches, using the bank microbenchmark at three contention levels. Crafty generally outperforms NV-HTM and DudeTM, especially under low contention and at low thread counts.
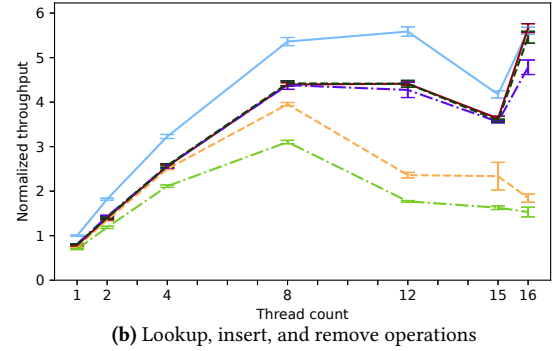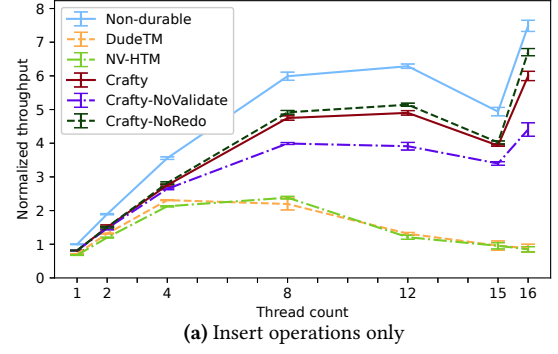

(a) Insert operations only

(b) Lookup, insert, and remove operations

Figure 7. Throughput of Crafty and competing approaches, on the B+ tree microbenchmark, for mixed operations and insert only. Crafty scales better than NV-HTM and DudeTM and has low overhead compared with Non-durable.

VALIDATE succeeds since there is no true contention. Results in our extended arXiv version [14] support this conclusion: Crafty-NoValidate incurs many explicit aborts at thread counts above 4, caused by failed timestamp checks.

B+ tree microbenchmark. Figure 7 shows the results for the B+ tree microbenchmark. NV-HTM and DudeTM scale poorly compared with Crafty and Non-durable, presumably as a result of serializing execution during transaction commit and when persisting writes; our extended results [14] do not show significant differences in transaction abort rates.

For both configurations of the benchmark at all thread counts, Crafty outperforms NV-HTM and DudeTM, and has modest overhead over Non-durable.

STAMP benchmarks. Figure 8 shows results for the STAMP benchmarks. Across the benchmarks, Crafty generally performs better than NV-HTM and DudeTM and scales as well as Non-durable (the exception is intruder, discussed below).

Figures 8(a) and 8(b) show that Crafty outperforms NV-HTM and DudeTM at thread counts above 4 for kmeans under both high and low contention.

Figures 8(c) and 8(d) show that Crafty adds modest overhead over Non-durable on vacation. Except for the high-contention vacation configuration above 8 threads, for which
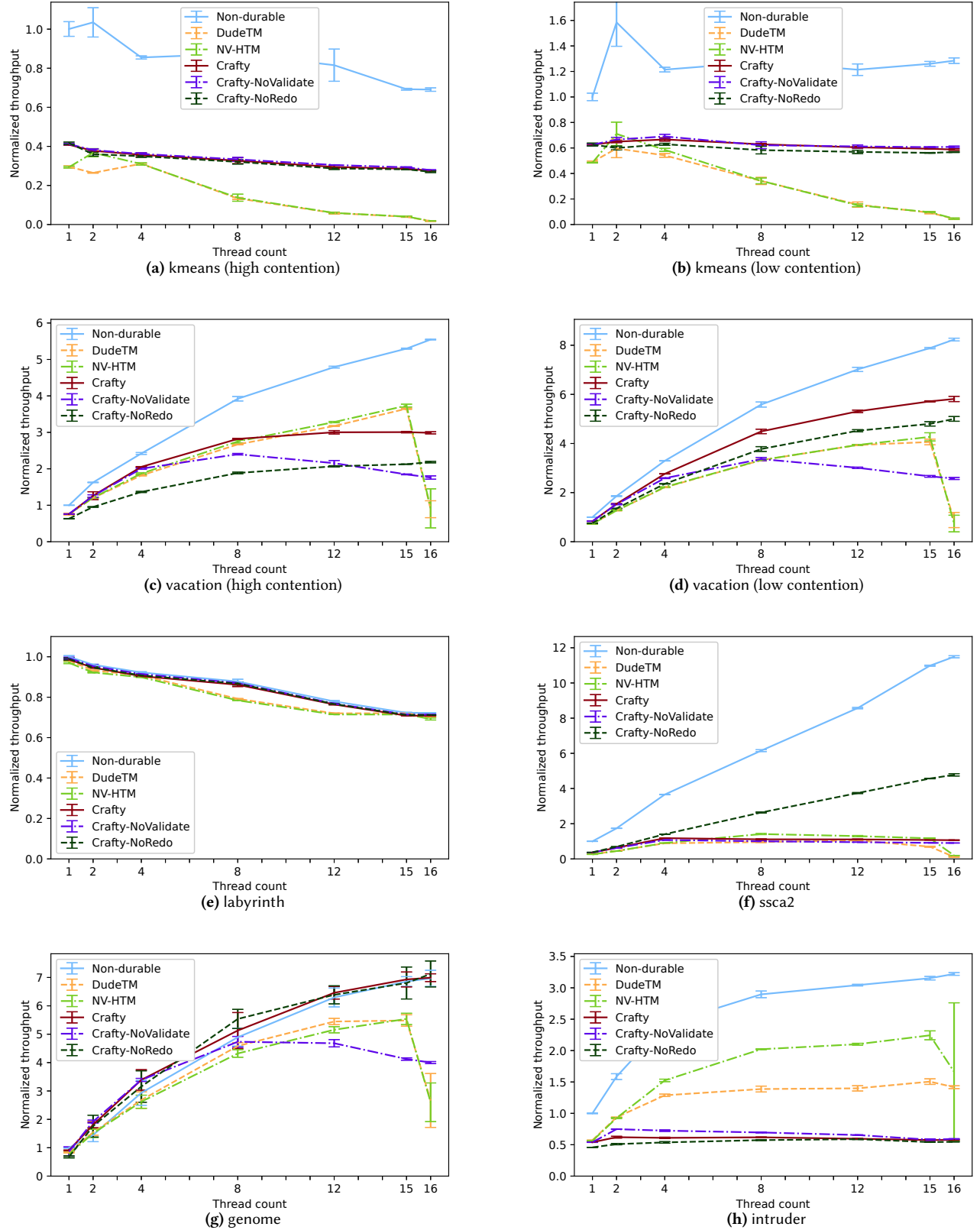
(a) kmeans (high contention)

(b) kmeans (low contention)

(c) vacation (high contention)

(d) vacation (low contention)

(e) labyrinth

(f) ssca2

(g) genome

(h) intruder

Figure 8. Throughput of Crafty and competing approaches, on the STAMP benchmarks. Crafty has low overhead and scales well at high thread counts.

NV-HTM and DudeTM perform best, Crafty outperforms competing approaches. (The sudden drop in NV-HTM and DudeTM's throughput at 16 threads is the same issue as described above for the bank microbenchmark.) Both figures show the benefits of using both the REDO and VALIDATE phases under higher thread counts but low contention.

Figures 8(e) and 8(f) show similar performance for Crafty, NV-HTM, and DudeTM on labyrinth and ssca2. An exception is that for ssca2, which has very low contention, Crafty-NoRedo performs significantly better than the other configurations. As our extended results show [14], across all thread counts, Crafty-NoRedo, which uses only LOG and VALIDATE phases, experiences very few aborts.

Figure 8(g) shows that Crafty scales well at high thread counts for genome, while NV-HTM and DudeTM are unable to scale quite as well. Crafty-NoValidate scales poorly with more threads, showing the value of the VALIDATE phase when the REDO phase fails frequently due to numerous simultaneous transactions.

Figure 8(h) shows that for intruder, Crafty performs worse than NV-HTM and DudeTM. While detailed statistics in our extended results [14] show that Crafty configurations commit and abort significantly more hardware transactions than the other configurations, these results do not explain Crafty's poor performance: Crafty inherently commits, and often aborts, more hardware transactions than other approaches across the other programs, yet generally outperforms NV-HTM and DudeTM. As of the camera-ready deadline, we have not been able to understand this issue better (we fixed an issue just days before the deadline that allowed our implementation to run intruder without error).

8 Related Work

Crafty leverages hardware transactional memory (HTM) to control persist ordering, while also supporting the use of commodity HTM for concurrency control in persistent transactions. To our knowledge, no prior work has used HTM to control persist ordering. Prior work supports commodity HTM for concurrency control in persistent transactions [7, 15, 39]. DudeTM and NV-HTM use shadow-paging-based copy-on-write mechanisms and incur scalability bottlenecks [7, 39]; we compared with them qualitatively and quantitatively in this paper. Giles et al. introduced an approach for HTM-based persistent transactions that requires instrumenting program reads [15], arguably forgoing a key benefit of using HTM instead of STM. In contrast with the prior work, which works around the challenges of combining persistence and HTM, Crafty *leverages* HTM to control persist ordering, as realized in the new nondestructive undo logging mechanism.

Modifying HTM. Several research efforts propose nontrivial modifications to commodity HTM to support persistent

transactions [3, 4, 16, 30, 54]. In contrast, Crafty shows how to leverage and work with contemporary systems.

Software persistent transactions. Many existing systems provide persistent transactions [9, 11, 13, 17, 19, 32, 33, 41, 43, 45, 47, 49, 52, 57]. These approaches use undo, redo, or copy-on-write mechanisms to provide failure atomicity. The approaches either assume that programs provide thread atomicity through locks or another concurrency control mechanism, or they apply STM to provide thread atomicity together with failure atomicity.

Failure atomicity of critical sections. Several approaches including *Atlas* provide failure atomicity for lock-based critical sections [5, 8, 23, 27, 40] or synchronization-free regions [18]. Crafty (in its thread-unsafe mode) could likewise provide failure atomicity for lock-based regions.

Failure ordering. This paper focuses on providing failure atomicity. Providing failure atomicity relies on the lower-level property of *failure ordering*, which refers to the order of persisted writes that the recovery observer sees. This paper's nondestructive undo logging leverages HTM to control persist ordering. Prior work introduces *memory persistency models*, which extend memory consistency models to incorporate the recovery observer [29, 31, 34, 44].

9 Conclusion

Nondestructive undo logging is a new crash-consistency mechanism that leverages commodity HTM to persist a transaction's undo log entries before its persistent writes. Crafty is a new design that uses nondestructive undo logging to provide persistent transactions. An evaluation shows that Crafty performs well compared with non-durable transactions and has better performance than state-of-the-art persistent transaction designs. These results show the potential for efficient persistent transactions using today's computing systems.

References

[1] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *VLDB* 11, 5 (2018), 553–565.

[2] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *VLDB* 10, 4 (2016), 337–348.

[3] Hillel Avni and Trevor Brown. 2016. Persistent Hybrid Transactional Memory for Databases. *VLDB* 10, 4 (2016), 409–420.

[4] Hillel Avni, Eliezer Levy, and Avi Mendelson. 2015. Hardware Transactions in Nonvolatile Memory. In *DISC*. 617–630.

[5] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-volatile Memory. In *ISMM*. 55–67.

[6] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*.

[7] Daniel Castro, Paolo Romano, and João Barreto. 2018. Hardware Transactional Memory Meets Memory Persistency. In *IPDPS*. 368–377.

[8] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *OOPSLA*. 433–452.

[9] 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. In *ASPLOS*. 105–118.

[10] Nachshon Cohen, Michal Friedman, and James R. Larus. 2017. Efficient Logging in Non-Volatile Memory by Exploiting Coherency Protocols. *PACMPL* 1, OOPSLA, Article 67 (2017), 24 pages.

[11] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *SPAA*. 271–282.

[12] Luke Dalessandro and Michael L. Scott. 2009. Strong Isolation is a Weak Idea. In *TRANSACT*.

[13] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *EuroSys*. 15:1–15:15.

[14] Kaan Genç, Michael D. Bond, and Harry Guoqing Xu. 2020. Crafty: Efficient, HTM-Compatible Persistent Transactions. arXiv:2004.00262 Extended version of PLDI 2020 paper.

[15] Ellis Giles, Kshitij Doshi, and Peter Varman. 2017. Continuous Checkpointing of HTM Transactions in NVM. In *ISMM*. 70–81.

[16] Ellis Giles, Kshitij Doshi, and Peter Varman. 2018. Hardware Transactional Persistent Memory. In *MEMSYS*. 190–205.

[17] Ellis R. Giles, Kshitij Doshi, and Peter Varman. 2015. SoftWrAP: A Lightweight Framework for Transactional Support of Storage Class Memory. In *MSST*. 1–14.

[18] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-Free Regions. In *PLDI*. 46–61.

[19] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *USENIX*. 913–928.

[20] Tim Harris and Keir Fraser. 2003. Language Support for Lightweight Transactions. In *OOPSLA*. 388–402.

[21] Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory* (2nd ed.). Morgan and Claypool Publishers.

[22] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*. 289–300.

[23] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications. In *EuroSys*. 468–482.

[24] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *USENIX*. 703–717.

[25] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *VLDB* 8, 4 (2014), 389–400.

[26] Intel Corporation. 2018. 3D XPoint[TM]: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[27] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *ASPLOS*. 427–442.

[28] Joseph, Izraelevitz, Lingxiang Xiang, and Michael L. Scott. 2017. Performance Improvement via Always-Abort HTM. In *PACT*. 79–90.

[29] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *MICRO*. 660–671.

[30] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *ISCA*. 452–465.

[31] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level persistency. In *ISCA*. 481–493.

[32] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *ASPLOS*. 399–411.

[33] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *ASPLOS*. 399–411.

[34] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. 2016. Delegated persist ordering. In *MICRO*. 1–13.

[35] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. 256–267.

[36] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *SOSP*. 460–477.

[37] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM* 21, 7 (1978), 558–565.

[38] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *ISCA*. 2–13.

[39] 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 *ASPLOS*. 329–343.

[40] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *MICRO*. 258–270.

[41] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. 2018. Persistent Memory Transactions. arXiv:1804.00701

[42] Jason Mars and Naveen Kumar. 2012. BlockChop: Dynamic Squash Elimination for Hybrid Processor Architecture. In *ISCA*. 536–547.

[43] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. In *EuroSys*. 499–512.

[44] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *ISCA*. 265–276.

[45] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak Persistency Semantics from the Ground up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. *PACMPL* 3, OOPSLA, Article 135 (2019).

[46] Ravi Rajwar and James R. Goodman. 2001. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO*. 294–305.

[47] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *DSN*. 151–163.

[48] Carl G. Ritson and Frederick R.M. Barnes. 2013. An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *CPA*. 271–292.

[49] Andy Rudoff. 2017. Persistent Memory Programming. *Login: The Usenix Magazine* (2017), 34–40. Issue 2.

[50] Andy M. Rudoff. 2016. Deprecating the PCOMMIT Instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[51] Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. 2017. Legato: End-to-End Bounded Region Serializability Using Commodity Hardware Transactional Memory. In *CGO*. 1–13.

[52] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS*. 91–104.

[53] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *PLDI*. 347–362.

[54] Zhaoguo Wang, Han Yi, Ran Liu, Mingkai Dong, and Haibo Chen. 2015. Persistent Transactional Memory. *CAL* 14 (2015), 58–61. Issue 1.

[55] H.-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil L Lee, Frederick T. Chen, and Ming-Jinn Tsai. 2012. Metal-Oxide RRAM. *Proc. IEEE* 100, 6 (2012), 1951–1970.

[56] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. 2010. Phase Change Memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[57] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *PLDI*.

[58] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *FAST*. 323–338.

[59] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *SOSP*. 478–496.

[60] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *SC*. 19:1–19:11.

[61] Hanbin Yoon, Justin Meza, Naveen Muralimanohar, Norman P. Jouppi, and Onur Mutlu. 2014. Efficient Data Mapping and Buffering Techniques for Multilevel Cell Phase-Change Memories. *TACO* 11, 4 (2014), 40:1–40:25.

[62] Pantea Zardoshti, Tingzhe Zhou, Yujie Liu, and Michael Spear. 2019. Optimizing Persistent Memory Transactions. In *PACT*. 219–231.

[63] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *ASPLOS*. 3–18.

[64] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *OSDI*. 461–476.