PHyTM: Persistent Hybrid Transactional Memory

Hillel Avni
Huawei Technologies
hillel.avni@huawei.com

Trevor Brown
University of Toronto
tabrown@cs.toronto.edu

Abstract

The availability of hardware transactional memory (HTM) and the feasibility of persistent hardware transactions make them a natural choice for in-memory database synchronization. However, limitations on the size of hardware transactions and the lack of progress guarantees by modern HTM implementations prevent some transactions from obtaining the benefit of hardware transactional memory. In this paper, we study persistent hybrid TM, which allows hardware assisted ACID transactions to execute concurrently with pure software transactions. This allows applications to gain the benefit of persistent HTM while accommodating unbounded transactions with a high degree of concurrency.

1. Introduction

Non-volatile memory (NVM) is an upcoming technology that promises to revolutionize computer memory. It is not currently commercially available, but manufacturers have developed prototypes, and have released performance information about these prototypes to the public. NVM is expected to become cheaper, faster and more power efficient than DRAM, and will likely become ubiquitous.

Researchers have just begun to understand how machines with NVM should be programmed. The programming model for NVM is still in flux, and several companies are competing to bring an implementation to market. In a system with NVM, the processor cache and registers are volatile, and writes to cache are asynchronously flushed to NVM (at any time, and without the programmer's knowledge). A programmer can also cause a cacheline to be flushed to NVM by invoking a primitive called *Flush*. Another primitive called a *persistence barrier* is provided to allow a thread to block until the cache line has been flushed to NVM. The key challenge in developing software for NVM is to ensure that the system is always left in a consistent state if a power failure occurs and the cache and registers are cleared.

Another recent technology called hardware transactional memory (HTM), which brings database-style transactions to shared memory, was recently implemented in Intel processors. Intel's implementation of HTM is best effort, which means that no transaction is ever guaranteed to commit. Thus, a non-transactional fallback path must be provided by a programmer to be executed if a transaction aborts sufficiently many times. The simplest fallback path simply reexecutes the body of a transaction after taking a global lock (that prevents other processes from performing transactions). However, this naive approach does not work with NVM.

The interplay between transactional memory and NVM proposals is particularly interesting, because transactions must appear to be atomic, but writes performed by the fallback path can be persisted (flushed to persistent memory) at any time. Therefore, the fallback path must be carefully designed to avoid exposing partial effects of an in-flight transaction to other processes in the event of a power failure. An additional complication arises from the fact that HTM cannot directly modify main memory. Any modifications to shared memory that are made by a transaction are performed on a copy of the data stored in the private cache of the core running the transaction. Thus, there is a timing window between when a transaction commits, and when the changes are flushed from cache into main memory, when a power failure could cause the results of a committed transaction to be lost.

Recent work by Avni et al. [2] introduced the first algorithm, *PHTM*, that allows hardware transactions to be performed in a system with NVM. At a high level, PHTM uses a redo log to ensure that no committed changes are lost. The authors propose a modification to Intel's HTM implementation that allows a single bit to be flushed to NVM atomically as part of a transactional commit. This bit allows them to simultaneously commit a transaction and flag a record of the redo log in NVM as *complete* so that, after a power failure, it will be replayed if and only if its transaction committed. The fallback path in PHTM is a software transactional memory (STM) called PSTM that was designed for use with NVM. Unfortunately, PSTM serializes all transactions, and the algorithm does not allow concurrency between hardware and software transactions. This eliminates all concurrency whenever a process is executing on the fallback path, and makes the algorithm unlikely to scale as the number of cores in HTM systems increases.

In the transactional memory (TM) literature, hybrid TM was introduced to solve similar performance issues. Hybrid TM algorithms improve performance by using STM algorithms that allow concurrency on the fallback path, and designing the fast path algorithm so that hardware and software transactions can run concurrently. However, existing hybrid TM algorithms do not work with NVM, so new algorithms are needed.

The main contribution of this work is *PHyTM*, the first hybrid TM for systems with NVM. PHyTM provides linearizable transactions with opacity and deadlock- and livelock-freedom. The strength of PHyTM lies in the high degree of concurrency that it offers. It uses a fast, hardware path and a slow, software path that can run concurrently. To avoid livelock, transactions on the slow path may occasionally take a global lock, which excludes other transactions *on the slow path*, but allows transactions on the fast path to continue. PHyTM allows transactions to continue to run in hardware, regardless of the actions of transactions on the slow path. This makes PHyTM an appealing option for in-memory databases, which demand persistence for ACID transactions, and often feature many large queries that must run in software.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

The rest of this paper is structured as follows. Section 2 gives a detailed description of our model. Since PHyTM builds upon the logging mechanism of PHTM, we use Section 3 to motivate and describe its implementation. We then describe the PHyTM algorithm in Section 4, and its implementation in Section 5. A formal proof of correctness and progress is given in Section 6. Related work is discussed in Section 7. Finally, we conclude in Section 8.

2. Model

We consider an asynchronous shared memory system with n threads (also called processes).

2.1 Memory

The memory is organized into a hierarchy, with the lowest level, *main memory*, consisting either entirely of NVM, or of a mixture of NVM and DRAM. This lowest level is logically divided into *cache lines* (which are typically 64 bytes on modern systems). The next levels of the hierarchy are *cache* levels, which contain copies of cache lines that appear in main memory. A cache coherence protocol ensures that threads see a consistent view of main memory despite the existence of multiple cached copies of some memory locations. At the highest level of the memory hierarchy are *registers*, special memory locations reserved in each processor for temporary computations. Generally, operations on objects lower in the memory hierarchy are orders of magnitude slower than operations on objects higher in the hierarchy. NVM is expected to be slower than DRAM for write operations, but at least as fast for read operations.

2.2 Failures

We assume that the system can experience power failures, which result in all contents of volatile memory being lost. DRAM is volatile, and so are all levels of the cache hierarchy and all registers. If the cache and registers in a system were, instead, made persistent, then no algorithms would require no changes to function on a system with NVM. However, the cache is likely to remain volatile for the foreseeable future due to hardware limitations (e.g., the relatively slow speed of writes and high failure rate of NVM compared to the memory currently used to implement the cache). After a power failure, only NVM still contains information.

System recovery after a power failure is performed by a single *recovery thread* which executes a special recovery procedure. The recovery procedure repairs the data structure *before* other threads resume execution. Since the recovery thread runs alone, it has considerable latitude to perform actions that would otherwise appear dangerous, such as forcefully releasing locks that were held by other threads before the crash.

2.3 Hardware transactional memory

We consider Intel's implementation of HTM. When a transaction loads a memory location, the cache coherence protocol loads the cache line that contains the memory location into the processor cache in *shared* mode. In this mode, other processors are also allowed to load this cache line into their caches. When a transaction writes to a memory location, its cache line is first set to *exclusive* mode, and any copies of the cache line in other processor caches are *invalidated* (deleted). Then, the write is performed in the cache.

Since hardware transactions do not directly modify main memory, any modified cachelines must be flushed to NVM at some point after a transaction has committed. This is accomplished using a hardware primitive called *FLUSH*, which takes a memory address *addr* as its argument. *FLUSH*(*addr*) causes the cache coherence protocol to flush the most up-to-date copy of the cache line that contains *addr* to main memory. *FLUSH*(*addr*) can cause any transactions that have *addr* in their read-sets to abort. If a power failure occurs while a transaction is in the middle of flushing its data to NVM, the system could be left in an inconsistent state, with part of a committed transaction recorded, and part of it lost.

We assume the same extension to the implementation of the HTM commit operation as Avni et al. [2]. An HTM transaction is committed using an instruction called tx_end_log, which takes the address of a single *logged* bit, as its argument. This bit is atomically set and flushed to NVM at the same time as the transaction is committed (in cache). We also assume their *transparent flush* operation *TFLUSH*, which takes a memory address *addr* as its argument. *TFLUSH*(*addr*) has the same effect as *FLUSH*(*addr*), except that it will not cause any transaction to abort.

3. Logging in PHTM

Since we build on the logging mechanism used for PHyTM from PHTM, we now expand upon the brief description of PHTM that was given in the introduction.

To eliminate the risk of losing data to a power failure, PHTM adds transaction logging. Traditional transaction logs have two major disadvantages: they can contain many transactions, and they store global information about the order in which transactions committed, so that a recovery process can decide what to do if, e.g., two threads write x=2 and x=3, respectively. These kinds of logs are very expensive to maintain, and offer more generality than is necessary for PHTM.

In order to limit the size of its logs, PHTM requires each thread to flush the results of its last transaction to NVM before starting another. This way, PHTM only needs to be able to recover one transaction for each process (namely, the current one). Consequently, PHTM only needs to store at most one transaction per thread in the log. PHTM is also able to log transactions without any ordering information, provided that the log never simultaneously contains two different writes to the same address. PHTM guarantees this property by having each transaction lock each addresses it will write, and hold this lock until its log is no longer necessary (and will no longer be used by a recovery thread). Holding locks until the log is no longer needed slightly lengthens the contention window of the transaction, and may cause a small amount of additional contention. (However, as we mentioned above, this allows PHTM to implement a more efficient log.)

4. PHyTM algorithm

In PHyTM, transactions can execute on an HTM-based fast path, and an STM-based slow path. To guarantee progress, transactions on the slow path must sometimes take a global lock. We begin by describing the slow path.

4.1 STM-based slow path

Lock-based STM algorithms feature two common locking methodologies: commit-time locking (CTL) and encounter-time locking (ETL). CTL makes it fairly easy to guarantee progress, because threads know the entire read-set before they begin locking: transactions can sort their read-sets and lock addresses in a consistent order, avoiding deadlock. However, CTL makes it difficult to satisfy the common correctness condition *opacity* [8, 9], which intuitively states that threads cannot observe the partial results of transactions. Transactional memory implementations that do not satisfy opacity can cause threads running transactions to enter infinite loops, encounter unexpected segmentation faults, etc., where it would be impossible to do so in a sequential execution. Since CTL does not acquire any locks until just before committing, all of the transaction's reads are necessarily done without taking locks. Without additional synchronization mechanisms, threads may read partial effects of other transactions, violating opacity.

Because of the difficulty of providing opacity with CTL, PHyTM uses ETL, which requires transactions to lock each address it will read *before* reading it. ETL makes it fairly straightforward to prove opacity. Since a thread can read an address only after locking it, and the thread will unlock its addresses only after performing all of its writes, no other transaction can see any of these writes until they have all been completed. Thus, ETL simplifies the proof of correctness for the STM. Unfortunately, it complicates progress: if transactions read addresses in different orders, then deadlock can occur. For example, if transactions T_1 is trying to read A and then B, and transaction T_2 is trying to read B and then A, each of them can lock one address, and then deadlock occurs.

We can avoid deadlock by using a *TryLock* primitive instead of a *Lock* primitive. Unlock Lock, which blocks until the lock is free, TryLock immediately returns false if the lock is held. In the aforementioned scenario, T_1 and T_2 will each return immediately from their second invocation of TryLock, and can then abort their transactions and try them again. However, the same scenario could arise over and over again, causing an infinite sequence of aborts. This is called *livelock*. As we explain below, we can avoid livelock by having threads on the slow path that are suspected of being livelocked take a global lock.

At a high level, PHyTM's STM path locks each address it encounters, performs all of its reads and logs its writes, then flushes its log to NVM before writing its values to memory and flushing them to NVM. For improved concurrency between readers, we use reader-writer locks, which can be acquired by either a single writer or multiple readers. The implementation must be careful to ensure that the log is atomically flushed to NVM, so that it will be replayed by the recovery thread, *precisely* when it is committed. (Otherwise, committed transactions might be lost, or transactions that have not yet been committed might be replayed by the recovery thread.)

4.2 HTM-based fast path

Like the slow path, the fast path acquires locks on all of the addresses it will write to, and then logs its writes. As we described above, this prevents the log from containing two writes to the same address. However, the fast path differs from the slow path in two crucial ways. First, the fast path actually performs its writes immediately after logging them (without waiting for the log to be replayed). This only works on the fast path, because these writes will all remain in the thread's private cache until the transaction commits. Second, the fast path does not acquire any locks when it *reads* addresses. Instead, it *reads the state of the lock* for these addresses. If the lock is currently locked by a writer (*write-locked*), then the transaction aborts. Reading the lock state causes the HTM transaction to *subscribe* to the lock, so that if is unlocked when the transaction first checks its state, but is locked by another thread at some later point before the transaction commits, then the transaction will abort.

At a high level, PHyTM's HTM path *subscribes* to locks for the addresses it reads, and locks the addresses it writes, logging and performing its writes as it locks each address to be written. When all of its writes are finished, it flushes its log to NVM and uses tx_end_log to atomically commit the transaction and mark its log as *completed*, so that a recovery thread will replay it, should a power failure occur. Finally, the HTM path replays its log entry, flushing all of its writes to NVM, and then clears its log entry. If a transaction fails sufficiently many times on the fast path, it moves to the slow path.

4.3 Optimizing fast path reads/writes

We can take advantage of HTM's ability to subscribe to a lock's state to implement an optimization. Suppose an HTM transaction reads a lock's state, but delays checking whether the lock is held until right before the transaction commits. For simplicity, assume that a lock's state is zero if it is not held by any thread, and is non-zero otherwise. Then, the states of all locks read by the transaction can be aggregated in a thread-local variable by using bitwise-OR, so that a single branch instruction performed just before committing is sufficient to check if any locks were held throughout the transaction.

Delaying these lock state checks can violate opacity within the transaction, since it effectively lets the transaction ignore locks until commit time. However, all of the problems that are created by violating opacity are resolved elegantly by Intel's HTM implementation, because it *sandboxes* transactions: Any exceptions, such as segmentation faults, that occur inside a transaction simply cause the it to abort[1].

The same optimization can also be applied to writes. Each time a hardware transaction tries to acquire a lock as a writer, rather than immediately checking whether it successfully acquired the lock (and, if not, aborting), it can simply save the lock state, and check the return value just before the transaction commits. As above, the lock states can be aggregated in a thread-local variable using bitwise-OR, and checked using a single branch instruction.

Path	Fast	Slow-R	Slow-W
Fast	Yes	Yes	Yes
Slow-R	Yes	Yes	No
Slow-W	Yes	No	No

Figure 1. Table showing which transactions can run concurrently. Slow-R (Slow-W) represents a transaction on the slow path holding the global lock as a reader (writer).

4.4 Ensuring progress with a global lock

As we mentioned above, the slow (STM) path uses a global reader/writer-lock to guarantee progress. Each transaction on the slow path starts with a budget for the number of times it can abort before it must take a global lock to ensure progress. Whenever a transaction is retried on the STM path, if it has not yet exhausted its budget, it acquires the global lock as a *reader*. Since many readers can acquire the lock, this allows concurrency with other threads on the slow path. However, if the transaction has exhausted its budget, it acquires the global lock as a *writer*, which blocks all other threads on the slow path. This makes it impossible for indefinite livelock to occur on the slow path, because, eventually, threads will exhaust their budgets and resort to taking the global lock as writers, which will serialize them.

Although a transaction that has acquired the global lock as a writer does not run concurrently with any other transaction on the slow path, it still acquires all of its per-address locks. This allows transactions on the HTM fast path to continue running concurrently with a single transaction that holds the global lock as a writer. Figure 1 summarizes whether transactions on each pair of paths can run concurrently.

5. PHyTM implementation

In this section, we give the full details of the PHyTM implementation. The fast and slow path each provide a set of operations for starting and committing transactions, and reading and writing memory locations. These functions are not directly called from user code. Instead, a user simply invokes primitives provided by the compiler for starting and committing transactions, and the compiler automatically instruments the user's code so that it executes transactions (on

[1] Infinite loops that occur because opacity is violated are slightly trickier. However, within a transaction, they *will* eventually terminate, because of an internal buffer overflow, an exception, or a context switch. Of course, if infinite loops are a significant concern, one can simply skip this optimization.

```
type thread_log_t
// data for write-set logging (persistent)
int wsize        // size of the write-set
word* wset[]     // addresses in the write-set
word wdata[]     // data to be written by the txn
bool logged      // true if the log entry is complete

// data for read-set (volatile)
int rsize        // size of the read-set
word* rset[]     // addresses in the read-set

// variables for control flow (volatile)
int attempts     // attempts on the current path
bool lockfail    // true if a lock-failure occurred

shared thread_log_t entries[]
shared rwlock_t locks[]
shared rwlock_t globallock
shared const int MAX_HTM_ATTEMPTS
shared const int MAX_STM_ATTEMPTS
```

Figure 2. Data structures for PHyTM

the appropriate path) using the functions we provide. We begin by describing the underlying data structures.

5.1 Data structures

The data structures for the PHyTM implementation appear in Figure 2. Broadly, they consist of per-thread log entries, locks to protect memory addresses, the global reader/writer-lock introduced in Section 4, and a pair of constants. The per-thread log entries are stored in an array, *entries*, which has one entry per thread. The locks used transactions to protect memory addresses are stored in an array called *locks*. (Note: on a real system, the *locks* and *entries* arrays must be padded to avoid false sharing.) The two constants, *MAX_HTM_ATTEMPTS* and *MAX_STM_ATTEMPTS*, store the maximum number of times a transaction can be attempted on the fast path and slow path, respectively.

5.1.1 Reducing the number of locks

To avoid the enormous space overhead of dedicating a unique lock to each memory address, we use a fixed number of locks, which are stored in an array called *locks*. These locks are accessed via a function, *GetLockAddr*, which hashes a memory address into the array of locks. Although mapping multiple addresses to the same lock dramatically reduces the space complexity of PHyTM (from half of all memory to an additive constant), it can cause false conflicts if threads simultaneously try acquire locks on two different addresses that map to the same lock. The same approach was taken by Dice et al. in possibly the most well known STM, *TL2* [6].

5.1.2 Per-thread log entries

We now describe the contents of the per-thread log entries. The variables in a log entry can be divided into three categories: variables for the write-set, for the read-set, and for managing the movement of transactions between paths.

The write-set is represented by four variables: *wsize*, *wset*, *wdata* and *logged*. *wsize* contains the number of addresses in the write-set. *wset* is an array that contains all of the addresses in the write-set. *wdata* in an array that contains the values written by the transaction to the addresses in the write-set. *logged* is a bit that is true if the log entry is *complete*, meaning that all of its data has been written and flushed to NVM by the thread performing the transaction. This bit indicates to the recovery thread that this log entry should be replayed, should a power failure occur.

The read-set is represented by two variables: *rsize* and *rset*, which are analogous to *wsize* and *wset*. Unlike the write-set, the read-set is not flushed to NVM, and is never used by the recovery thread. The read-set is only used by transactions on the slow path, which use it simply to keep track of which addresses they've locked as readers.

The variables used to manage the movement of transactions between paths are *attempts* and *lockfail*. *attempts* stores the number of attempts that the transaction has made on the current path. On the fast path, the transaction will move to the slow path once this number reaches *MAX_HTM_ATTEMPTS*. On the slow path, the transaction will acquire the global lock as a writer once this number reaches *MAX_STM_ATTEMPTS*. *lockfail* is a thread-local temporary variable used to store the bitwise-OR of the lock states to implement the optimization described in Section 4.3. Neither of these variables are flushed to NVM.

```
void STM_Begin(thread_log_t* rec)
  rec->attempts++
  if rec->attempts <= MAX_STM_ATTEMPTS then
    ReadLock(globallock)
  else
    WriteLock(globallock)

word STM_Read(word* addr, thread_log_t* rec)
  rwlock_t* lock = GetLockAddr(addr)
  if !TryReadLock(lock) then
    ResetLogEntry(rec)
    Unlock all locks (including globallock)
    retry the transaction
  val = *addr

  // add addr to read-set (so it is unlocked later)
  rec->rset[rec->rsize] = addr
  rec->rsize++
  return val

// precondition: STM_Read(addr, rec) has previously
     been invoked in this transaction
void STM_Write(word* addr, word val, thread_log_t* rec)
  rwlock_t* lock = GetLockAddr(addr)
  if !TryWriteLock(lock) then
    ResetLogEntry(rec)
    Unlock all locks (including globallock)
    retry the transaction

  // add <addr, val> to the write-log
  rec->addr[rec->wsize] = addr
  rec->wdata[rec->wsize] = val
  rec->wsize++

bool STM_Finalize(thread_log_t* rec)
  // transparently flush the log entry
  FlushLog(rec)

  // log entry is ready to be replayed
  rec->logged = 1
  TFLUSH(rec->logged)

  // replay the log entry to perform & flush all writes
  ReplayLogEntry(rec, true /* perform writes */)
  Unlock all locks (including globallock)

  ResetLogEntry(rec)
  rec->attempts = 0
  return true
```

Figure 3. Operations for the STM path

5.2 The slow path

The slow path provides four operations: *STM_Begin*, which starts a transaction, *STM_Read*, which replaces a standard read from memory, *STM_Write*, which replaces a standard write, and *STM_Finalize*, which commits a transaction. The pseudocode for the slow path operations appears in Figure 3.

An *STM_Begin* operation increments the number of transactional *attempts* stored in the transaction's log entry, and acquires the global

lock as a reader (if the tranasction's budget for attempts has not yet been exhausted) or a writer (if it has).

An *STM_Read* operation acquires a read-lock, reads the address, and saves it in its read-set. An *STM_Write* operation acquires a write-lock, which serves two purposes. This lock grants exclusive access to the address being written, and exclusive permission to store that address in its write-log entry. The *STM_Write* then adds the address and the value to be written to its write-log entry (but does not yet make any effort to flush it to NVM[2]). If *STM_Read* or *STM_Write* fails to acquire a lock, the transaction is aborted, all locks are released, and the transaction is retried from scratch.

To commit an STM transaction, a thread invokes *STM_Finalize*. *STM_Finalize* flushes the write-log entry to NVM, and then sets and flushes a logged bit in the log entry, which indicates that it is ready to be replayed by the recovery thread if a power failure occurs. The transaction is *committed* precisely when the logged bit reaches NVM. Next, *STM_Finalize* invokes a function called *ReplayLogEntry* (which appears in Figure 5) to replay its own log entry, performing all of its writes and flushing them to NVM. *ReplayLogEntry* also clears and flushes the logged bit to indicate that the log entry no longer needs to be replayed. Finally, *STM_Finalize* unlocks all of its locks and prepares its log entry for reuse by the thread's next transaction.

5.3 The fast path

The fast path provides the same operations as the slow path, but their names have an "HTM" prefix instead of "STM." The implementation of these operations appears in Figure 4.

An *HTM_Begin* operation first increments the number of transactional *attempts* stored in the transaction's log entry. If the transaction has exceeded its budget of attempts on the fast path, the thread resets the *attempts* counter, and moves the transaction to the software path.

An *HTM_Read* operation reads the lock state for the address being read, and saves this state in the *lockfail* variable of the log entry, as per the optimization described in Section 4.3. Then, it reads and returns the address. An *HTM_Write* operation tries to lock the address being written as a writer, and saves the resulting lock state in the *lockfail* variable of the log entry, using the aforementioned optimization. This lock grants exclusive access to the address being written, and exclusive permission to store that address in its write-log entry. Next, *HTM_Write* adds the address and the value to be written to the transaction's write-log entry. Finally, it performs its actual write.

To commit a transaction on the fast path, a thread invokes *HTM_Finalize*. *HTM_Finalize* begins by checking the *lockfail* variable in the transaction's log entry at line 106. If *HTM_Read* sees an address that is already locked by a writer at line 80, or *HTM_Write* fails to acquire a lock at line 86, then the *lockfail* variable will be non-zero, and the transaction will abort. This ensures that transactions on the fast and slow paths do not interfere with one another. Next, *HTM_Finalize* flushes the write-log entry to NVM, and then executes a tx_end_log instruction at line 106. This instruction simultaneously commits the transaction, and sets and flushes the *logged* bit in the log entry (indicating that the log entry is ready to be replayed by the recovery thread if a power failure occurs). The transaction is *committed* precisely when this instruction is executed. After this, *HTM_Finalize* invokes *ReplayLogEntry* (see Figure 5) to replay its own log entry, flushing its writes to NVM. *ReplayLogEntry* also clears and flushes the logged bit to indicate that the log entry no longer needs to be replayed. (Unlike the invocation of *ReplayLogEntry* in *STM_Finalize*, this invocation

[2] In our model, the contents of the write-log entry may be flushed to NVM automatically at any time by the hardware. Here, we are simply remarking that the thread does not explicitly flush its modifications to its log entry, yet.

```
69  void HTM_Begin(thread_log_t* rec)
70    rec->attempts++
71    if rec->attempts <= MAX_HTM_ATTEMPTS then
72      start hardware transaction
73    else
74      rec->attempts = 0
75      goto STM path
76
77  word HTM_Read(word* addr, thread_log_t* rec)
78    // check if addr is write-locked
79    rwlock_t* lock = GetLockAddr(addr)
80    rec->lockfail = rec->lockfail | IsWriteLocked(lock)
81    return *addr
82
83  void HTM_Write(word* addr, word val, thread_log_t* rec)
84    // try to write-lock addr
85    rwlock_t* lock = GetLockAddr(addr)
86    bool succmark = TryWriteLock(lock)
87    rec->lockfail = rec->lockfail | !succmark
88
89    // add <addr, val> to the write-log
90    int wsize = rec->wsize
91    rec->wset[wsize] = addr
92    rec->wdata[wsize] = val
93    rec->wsize++
94    // perform the write
95    *addr = val
96
97  void HTM_Finalize(thread_log_t* rec)
98    // transaction experienced a lock-failure
99    if rec->lockfail then abort transaction and retry
100
101   // transparently flush the log entry
102   FlushLog(rec)
103   // atomically commit (to cache) and
104   // simultaneously set rec->logged in NVM
105   // to indicate the log entry is ready to be replayed
106   tx_end_log(rec->logged)
107
108   // replay the log entry to flush all writes
109   ReplayLogEntry(rec, false)
110   Unlock all locks
111
112   ResetLogEntry(rec)
113   rec->attempts = 0
```

Figure 4. Operations for the HTM fast path

of *ReplayLogEntry* does not need to perform the transaction's writes, since they are already performed as part of the hardware transaction.) Finally, *HTM_Finalize* unlocks all of its locks and prepares its log entry for reuse by the thread's next transaction.

5.4 Recovery

After a power failure, the recovery thread runs a simple procedure called *Recovery* (see Figure 5). Locks are not flushed explicitly to NVM, but some of them may have been flushed to NVM automatically by the hardware, and they have to be released before threads can resume normal operation. So, *Recovery* begins by unlocking all threads' locks. (The recovery thread has the freedom to do this, because it is running alone in the system.) Next, it invokes *ReplayLogEntry* for each log entry in the log. This is the same procedure that is used by *STM_Finalize* and *HTM_Finalize* to complete a transaction once its log entry is flushed.

ReplayLogEntry first checks if the log entry has its *logged* bit set. If so, the transaction has been committed, and it must be persisted in NVM. Next, the transaction's writes are performed at line 138 (because $doWrites = true$ when *ReplayLogEntry* is invoked by *Recovery*). (Note that the recovery thread performs these (apparently redundant) writes even for hardware transactions, despite the fact that *HTM_Finalize* invokes *ReplayLogEntry* with $doWrites = false$, and does not perform these writes. Here, these writes are necessary, because after a hardware transaction commits, but before its writes

```
114 void ResetLogEntry(thread_log_t* rec)
115   // prepare log entry for the next txn attempt
116   rec->lockfail = 0
117   rec->wsize = 0
118   rec->rsize = 0
119
120 void FlushLog(thread_log_t* rec)
121   TFLUSH(rec->wsize)
122   int wsize = rec->wsize
123   for i = 1..wsize
124     TFLUSH(rec->wset[i])
125     TFLUSH(rec->wdata[i])
126
127 void Recovery(int nthreads)
128   Unlock all locks for all threads
129   for i = 1..n
130     ReplayLogEntry(entries[i], true)
131
132 void ReplayLogEntry(thread_log_t* rec, bool doWrites)
133   if rec->logged then
134     int wsize = rec->wsize
135     if doWrites then
136       // perform all writes
137       for i = 1..wsize
138         *rec->wset[i] = rec->wdata[i])
139
140     // transparently flush all writes
141     for i = 1..wsize
142       TFLUSH(*rec->wset[i])
143     // the log entry no longer needs replaying
144     rec->logged = 0
145     TFLUSH(rec->logged)
```

Figure 5. Functions common to HTM and STM paths

are flushed to NVM, they may be lost to a power failure.) We briefly argue that, when the power failure occurred, the thread performing the transaction held locks on all of the addresses in the transaction's write-set (so it is correct to perform these writes). Observe that the log entry's *logged* bit reset to zero at the end of *ReplayLogEntry*. It follows that a power failure occurred before the thread that was running this transaction could finish its invocation of ReplayLogEntry at line 63 in *STM_Finalize*, or line 109 in *HTM_Finalize*. In either case, the thread still held locks on all of the addresses in the transaction's write-set. *ReplayLogEntry* concludes by flushing all of the writes to NVM, and setting the *logged* bit to zero and flushing it to NVM.

5.5 Optimizing with non-transactional reads

In this section, we describe an optimization to PHyTM that can be applied in HTM system that allow threads to perform non-transactional reads and writes while inside a transaction. Note that the *HTM_Read* function reads both the value stored at an address and the state of its lock. A paper by Riegel et al. [16] observed that it is sufficient to subscribe only to the lock (which is acquired by both HTM and STM), and use a *non-transactional read* for the data itself. In PHyTM, this optimization is quite natural, since locks are already taken by HTM for logging, and not simply because of its interactions with the STM-based slow path. PHyTM can also use non-transactional reads and writes to maintain its redo-log, which is local to a single thread (with the exception of the recovery process, which runs alone in the system).

6. Correctness

In this section, we formally prove that the PHyTM algorithm implements transactional memory with opacity and linearizable transactions, and that the algorithm is deadlock- and livelock-free.

6.1 Progress

It is straightforward to *informally* argue that the algorithm is deadlock- and livelock-free. The algorithm is deadlock-free because it uses a non-blocking *TryLock* primitive for all locks except the global reader/writer-lock (which cannot cause deadlock), and releases all locks and restarts the transaction whenever an invocation of *TryLock* sees that a lock is already held. To show livelock-freedom, we suppose that transactions stop committing after some point in time, and obtain a contradiction. Intuitively, threads will eventually exhaust their budgets for transactional attempts on the fast path and slow path, and will all take the global lock as writers, at which point one of them will acquire the lock and commit a transaction, yielding a contradiction. However, the formal proof is deceptively subtle.

Definition 1. *HTM_Begin, HTM_Read, HTM_Write and HTM_Finalize are **fast-path operations**. STM_Begin, STM_Read, STM_Write and STM_Finalize are **slow-path operations**. Collectively, these are referred to as **PHyTM operations**.*

Lemma 1. *Every PHyTM operation is wait-free (terminates after a finite number of steps), except for STM_Begin.*

Proof. Recall that we assumed *TryReadLock* and *TryWriteLock* return immediately if the lock is held. Because of this assumption, *STM_Read*, *STM_Write*, *HTM_Read*, *HTM_Write* and *HTM_Begin* are straightline code. *HTM_Finalize* and *STM_Finalize* are straightline code, apart from their invocations of *FlushLog* and *ReplayLogEntry*. *FlushLog* and *ReplayLogEntry* are straightline code except for their loops, which each perform k iterations, where k is the value of *wsize* in the transaction's log entry. It is easy to verify that *wsize* is always positive and finite, since the only places it is modified are at line 117 in *ResetLogEntry*, where it is set to zero, and at line 52 in *HTM_Write* and line 52 in *STM_Write*, where it is incremented exactly once per write in the transaction. □

Theorem 2. *PHyTM is deadlock- and livelock-free. (Formally, if all transactions have finite read- and write-sets, and all threads take steps infinitely often, then transactions commit infinitely often.)*

Proof. Suppose not, to obtain a contradiction. Then, in some execution, after time t, all threads take steps infinitely often, but no transaction commits.

Claim 1. Eventually, every thread invokes only slow-path operations.

Suppose not, to obtain a contradiction. Then, some thread p executes fast-path operations infinitely often. Since no transactions can commit after time t, p must perform infinitely many fast-path operations without ever executing line 113. Furthermore, p cannot execute line 74, since it would then begin executing on the slow path, never to return to the fast path (and never again to execute a fast-path operation). Therefore, the *attempts* counter in the log entry is never reset to zero. Since transactions have finite read- and write-sets, p's transaction must abort infinitely often, which means p must invoke *HTM_Begin* for its transaction infinitely often, incrementing *attempts* each time. However, each transaction has a finite budget *MAX_HTM_ATTEMPTS* for attempts on the fast path, so p must eventually move to the slow path and stop executing fast-path operations, which is a contradiction.

Claim 2. Eventually, every thread either invokes *STM_Begin* infinitely often, or spins forever at line 24 or line 26 in *STM_Begin*.

Suppose not, to obtain a contradiction. Then, some thread p invokes *STM_Begin* finitely many times, and does not spin forever in *STM_Begin* (the only place in the code where unbounded spinning occurs). Since all other PHyTM operations are wait-free, and transactions have finite read- and write-sets, p must successfully commit a transaction after t, which is a contradiction.

We can now prove the theorem. Let σ be the set of threads that invoke *STM_Begin* infinitely often. Since the global lock used by PHyTM is deadlock-free, it is impossible for *every* thread to spin forever in *STM_Begin*. Thus, Claim 2 implies that σ is non-empty.

By a similar argument to the proof of Claim 1, every thread in σ eventually exhausts its budget of transactional attempts on the slow path. Therefore, eventually, every invocation of *STM_Begin* by a thread in σ attempts to acquire the global lock as a writer at line 26. Since the global lock is deadlock-free, eventually some thread in σ will successfully acquire the lock as a writer, at which point it will run alone on the STM path, so its transaction will be guaranteed to commit (which is a contradiction). □

6.2 Linearizability and opacity

In this section, we show that PHyTM transactions are linearizable, they satisfy opacity on the slow path, and they are sandboxed on the fast path so that they are not susceptible to the problems that can arise when opacity is not satisfied. In this section, we assume no power failures occur. (In the next section, we consider how they change things.)

Definition 2. *A **transaction attempt** by a thread p is any interval starting with an HTM_Begin (STM_Begin) by p and ending with the next HTM_Finalize (STM_Finalize) by p.*

In the course of trying to perform a transaction, a process may make several transaction attempts. One can think of a transaction as a collection transaction attempts[3].

Definition 3. *A transaction attempt on the fast path **commits** at its execution of tx_end_log at line 106. A transaction attempt on the slow path **commits** at its execution of TFLUSH at line 60.*

We now give the linearization points for committed and aborted transactions on the fast- and slow-path.

Linearization points

- Each committed transaction attempt on the fast path is linearized at its execution of tx_end_log at line 106 of *HTM_Finalize*.
- Each committed transaction attempt on the slow path is linearized at its TFLUSH instruction at line 60 of *STM_Finalize*.
- Each aborted transaction attempt on the fast path is linearized at its last execution of line 80 in *HTM_Read* or line 86 in *HTM_Write* **before** the *lockfail* bit was set.
- Each aborted transaction attempt on the slow path is linearized at its last execution of line 30 in *STM_Read* or line 44 in *STM_Write*.

Observe that committed transactions are linearized precisely at the moment they are committed.

Lemma 3. *Suppose a transaction attempt T changes an address $addr$ in main memory that is not a lock or part of a log entry. Then, the following statements hold.*

1. *T must commit.*
2. *T adds $addr$ to its write-log entry at lines 50-52 or lines 91-93.*
3. *T locks $addr$ as a writer at line 44 or line 86, before adding $addr$ to its write-log entry, and before writing to $addr$. T continuously holds this lock until after it commits.*

Proof. Suppose T is a software transaction. Then T must write to $addr$ at line 138 of *ReplayLogEntry*. Prior to invoking *ReplayLogEntry* at line 63, it commits at line 60 (proving Claim 1). Claim 2 and Claim 3 are immediate from the code.

Now, suppose T is a hardware transaction. Trivially, T must commit in order to change main memory (Claim 1). Claim 2 is immediate from the code. From the code, T must write to $addr$ at line 95. Just prior to this, T tries to lock $addr$ as a writer at line 86. If T succeeds, then Claim 3 is immediate from the code. However, if T fails to acquire the lock, then the log entry's *lockfail* bit is set, and the transaction will abort at line 99, which is a contradiction. Therefore, this case is impossible, and Claim 3 is proved. □

Lemma 4. *Every invocation of STM_Read on an address $addr$ returns the most recent value written to $addr$ by a committed transaction.*

Proof. Let R be an invocation of *STM_Read* on an address $addr$, r be the execution of line 34 in R, T be the last transaction with $addr$ in its write-set that committed prior to r, and v be the value that T writes to $addr$. Our goal is to prove that $addr$ contains v when step r is executed.

By Lemma 3, and a simple inspection of the code, T's write to $addr$ is flushed to main memory at some time t after it commits, but before it releases its locks (at the very latest, in *ReplayLogEntry*, but possibly earlier if the system automatically flushes the write). Since *STM_Read* sees that $addr$ is not locked at line 30 just before step r, we know that r happens after tine t. Thus, $addr$ contains v at time t, after T commits, but before r.

We argue that $addr$ does not change between t and r. Suppose, to obtain a contradiction, that $addr$ changes between t and r. Since $addr$ can be changed only by a committed transaction, and only while it holds writer-locks on every address in its write-set, if $addr$ is changed between t and r, another committed transcaction T' must hold a write-lock on $addr$ at some point between t and r. Since T and T' cannot both hold write-locks on $addr$ at the same time, and $addr$ can only be changed while it is write-locked, T' must have locked $addr$ *after* T. However, T' must commit while it holds a write-lock on $addr$, so T' must commit (and be linearized) after T, which is a contradiction. □

Corollary 5. *Every invocation of HTM_Read made by a transaction prior to its linearization point returns the most recent value written to $addr$ by a committed transaction*

Proof. For committed transactions, *HTM_Read* subscribes to the state of the lock for each address it reads, and sees that the lock is not held. Thus, the value it reads at line 81 is identical to value it would read if it had explicitly acquired a read-lock on the address. Consequently, the proof is identical to the proof of Lemma 4.

Aborted transactions are linearized at the first point where it sees a lock is already held by another thread at line 80 or fails to acquire a lock at line 86. Therefore, for every read prior to the linearization point, the transaction subscribes to the state of the lock at line 81, and sees that the lock is not held. From this point onward, the argument is the same as above. □

Theorem 6. *PHyTM provides opacity on the slow path, and sandboxes transactions on the fast path.*

Proof. Immediate from Lemma 4, Lemma 5 and the fact that fast path transactions are automatically sandboxed by the implementation of HTM. □

6.3 Recovery

In this section, we prove the correctness of the *Recovery* procedure that is invoked by the recovery thread after a power failure. Intuitively, this entails showing that the log is always well formed, and that no committed transactions are lost to a power failure.

Definition 4. *A transaction attempt is **logged** whenever the logged bit in its log entry is set.*

[3] Formally, a transaction by a thread p is the collection of transaction attempts by p, where the first transaction attempts begins when p's log entry contains *attempts* $= 0$, and the last transaction attempt ends when p's log entry *next* contains *attempts* $= 0$.

Observe that a committed transaction attempt is linearized, and becomes committed and logged at precisely the moment that their *logged* bit is flushed to NVM (so that they will be replayed by the recovery thread if a power failure occurs).

Lemma 7. *At all times, the set of log entries that have their logged bits set contains at most one instance of each memory address.*

Proof. By inspection of the code, a transaction attempt can be *logged* only while it holds all locks in its write-set. □

Lemma 8. *Every transaction attempt that commits before a power failure either terminates (meaning its invocation of HTM_Finalize or STM_Finalize terminates) prior to the power failure, or it is logged.*

Proof. Let T be a transaction that commits (and, consequently, is linearized) before a power failure. Suppose T does not terminates prior to the power failure. Then, since T commits before the power failure (and transactions commit and are logged at precisely the same time), T is logged before the power failure. □

Theorem 9. *Immediately after the recovery thread finishes executing the Recovery procedure, the contents of shared memory are exactly what they would be if all transaction attempts that had committed (and linearized) but not yet terminated when the power failure occurred had actually run to completion.*

Proof. Let T be a transaction attempt that committed but had not yet terminated when the power failure occurred. Since T committed before the power failure, it is logged, and it held write-locks on all addresses in its write-set when the power failure occurred. So, if T ran to completion, then it would have performed all of its writes and flushed them to NVM. Since T is logged, the *Recovery* procedure will perform all of its writes.

It remains to prove that the transactions whose log entries are replayed by the *Recovery* procedure will not interfere with one another. Since all of the transactions whose log entries will be replayed by the *Recovery* procedure are logged, Lemma 7 implies that all logged transactions operate on disjoint write-sets. □

7. Related work

Non-volatile RAM is expected to replace DRAM, either partially or entirely, as main memory [13]. There are already working prototypes of NVM such as phase-change memory (PCM) [13], spin-torque-transfer RAM (STT-RAM) [11], and memristors [17], and the new Intel architecture added special instructions (CLFLUSHOPT, PCOMMIT) [1] to access data in NVM. As memory becomes persistent, it is natural to make persistent transactional memory, i.e. to support full ACID TM transactions.

NV-Heaps [3] and Mnemosyne [18] are full system solutions. They include allocating persistent memory to applications, defining non-volatile variables in the compiler, and preventing illegal states such as a persistent object pointing to a volatile one. As part of their NVM support, they also provide persistent STM.

NV-Heaps includes an object-based persistent STM. It provides transactional objects, which can be opened for writing. Once an STM transaction T opens an object for writing, T copies the object to an undo log, and locks it. NV-Heaps maintain a volatile read log and a non-volatile undo log for each transaction. If a power failure occurs, any transactions in progress are aborted, and the undo log, which is persistent, is used to reverse any changes they made.

Mnemosyne persistent STM [18], which was published at the same time as NV-Heaps, is word-based, and is derived from TinySTM [7]. Mnemosyne buffers writes to avoid the maintenance of an undo log, and to work around the fact that writes can be flushed to NVM at any time. Buffering writes results in slower commits. Mnemosyne logs writes in per-thread redo logs, and logged writes are totally ordered by a global clock, which is taken from TinySTM.

Unfortunately, these software-based algorithms exhibit poor performance due to bookkeeping overhead and/or poor scalability due to locking serialization. Thus database transactions use fine-grained locking and no commercial database uses STM. Some database implementations [14, 19] use HTM for synchronization. However, these databases still use flush data to disk to achieve persistence.

PHTM [2] is a persistent version of HTM for machines that provide NVM. It writes a persistent bit in NVM at HTM commit-time, and logs the write-set in NVM, so that, at any time, if a power or hardware failure occurs, then the contents of shared memory can be recovered. PHTM executes reads at hardware speed, and its commits are instantaneously persistent at commit-time. PHTM provides both synchronization and durability for an in-memory database, but it carries the limitations of best effort HTM, and cannot commit large transactions (except sequentially, on a fallback path). PHyTM improves on PHTM by offering both persistent HTM and highly concurrent STM, to gain the performance benefit of HTM while maintaining parallelism when a transaction must execute in software.

The limitations of HTM were mentioned already in the seminal paper of Herlihy and Moss [10], but the first algorithms that allow a fast path concurrency with the slow path were introduced in 2006 [5, 12]. Since then, research on hybrid TM algorithms has been focused on optimizations to improve performance.

Optimizations for hybrid TMs have progressed in two directions:

- Reducing overhead by letting the slow path take a global sequential lock, which is sampled by the fast path on each access, in the HyNORec algorithms [4]. (In this direction, HTM is also used in the commit phase of an STM transaction, which eliminates the need for HTM to sample locks [15].)
- Attaching a versioned lock or a traditional lock to each address, which is sampled on each HTM read, and acquired by both paths for each write, in the HyLSA algorithm [16]. This algorithm greatly increases the size of HTM transactions, because locks must be sampled for each HTM read. However, this problem can be mitigated with the use of non-transactional reads and writes.

The NORec family of algorithms has lower overhead, but is less scalable, so we chose to pursue the same direction as HyLSA for PHyTM. Reads of data in PHyTM can be non-transactional. Since both persistent HTM and STM transactions acquire locks on each address in their write-sets (to ensure that no other transaction can write to these addresses until the current transaction's changes are flushed), it is sufficient for each transaction to subscribe to the state of the lock for each address (instead of subscribing both to the lock state and to the address it protects). Since each STM read also acquires a lock, separating the STM read-lock from the write-lock may also reduce unnecessary aborts caused by conflicts between STM reads and HTM reads.

8. Conclusion

Efficient, persistent hybrid TM will allow in-memory databases to benefit from the research accumulated in the TM literature. More than two decades ago, transactional memory started as a hardware proposal for efficient execution of short transactions, and, later, expanded to efficient synchronization of general transactions in memory. Recently, databases have begun to move away from disks and become fully in-memory. PHyTM's line of research promises to connect transactional memory with cutting-edge in-memory databases.

References

[1] Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[2] H. Avni, E. Levy, and A. Mendelson. Hardware transactions in nonvolatile memory. In *Proceedings of 29th International Symposium, DISC 2015*, pages 617–630. Springer, 2015.

[3] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[4] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: a case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 39–52, 2011.

[5] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 336–346, 2006.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In S. Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2006.

[7] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246. ACM, 2008.

[8] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.

[9] T. Harris, J. Larus, and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[11] Y. Huai. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS Bulletin*, 18(6), 2008.

[12] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. D. Nguyen. Hybrid transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006*, pages 209–220, 2006.

[13] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143, 2010.

[14] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 580–591, 2014.

[15] A. Matveev and N. Shavit. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 59–71, New York, NY, USA, 2015. ACM.

[16] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: the importance of nonspeculative operations. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 53–64, 2011.

[17] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, May 2008.

[18] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not.*, 47(4):91–104, Mar. 2011.

[19] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 26:1–26:15, New York, NY, USA, 2014. ACM.