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 applications 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. Our experiments demonstrate that PHyTM is fast and scalable for realistic workloads.

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. Writes to processor 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 process to block until the cache line has been flushed to NVM.

There is significant controversy over whether processor cache and registers will be volatile or non-volatile. Some researchers are investigating ways to provide enough residual power to flush this data to NVM in the event of a power failure [18]. This approach could allow applications to avoid any runtime overhead associated with providing persistence. However, hardware designers are skeptical about its feasibility, citing concerns about the amount of energy necessary to flush processor cache (since the cache is very large, and processors are complex and power-hungry). They suggest that future hardware will only use residual energy to flush data in volatile buffers on NVM-controllers [13]. Operating under this assumption, Intel is currently designing new and efficient flush instructions (CLWB and PCOMMIT) with NVM in mind [1]. These new instructions assume volatile caches, and allow the flushed data to stay in cache, to avoid cache misses. They also accelerate flushes to NVM.

We consider a system in which the processor cache and registers are volatile. In such a system, the key challenge is to ensure that NVM 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. (HTM has also been implemented in production systems by IBM, and in various research systems. We focus on Intel's implementation.) 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 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 processor 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 an 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. Its 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 PHTM 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 parallelism 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. Like PHTM, PHyTM uses redo logging to facilitate recovery after power failures. PHyTM provides linearizable transactions with deadlock- and livelock-freedom. It uses three execution paths: *Fast HTM*, which has uninstrumented reads, *Slow HTM*, which has instrumented reads and writes, and *STM*, which locks its read- and write-sets, and buffers all writes until its **write-back** phase, which happens at commit time. To avoid livelock, transactions on the STM path may occasionally take a global lock, which excludes other transactions *on the STM path*, but allows hardware transactions to continue.

Fast HTM and Slow HTM can run concurrently (because both use HTM, so conflicts are resolved by the hardware), and so can Slow HTM and STM (because Slow HTM acquires locks, just like STM). However, since Fast HTM has uninstrumented reads, it cannot run concurrently with STM without an additional mechanism to determine whether STM has left memory in an inconsistent state. Thus, each transaction T on Fast HTM subscribes to a counter that contains the number of transactions on the STM path that are in their write-back phases. If the counter is ever non-zero during T, then T may have seen inconsistent state, so it aborts. If T aborts sufficiently many times on Fast HTM because the counter is non-zero, then it moves to Slow HTM, where it can run concurrently with any transaction. If T aborts sufficiently many times for any other reasons, it moves to the STM path to guarantee progress.

In-memory databases demand persistence for ACID transactions, and their workloads often feature many large queries that exceed the capacity limitations of hardware transactions. These requirements make PHyTM an appealing option, since it allows transactions to commit in hardware concurrently with software transactions, and its uninstrumented reads on Fast HTM facilitate highly efficient queries.

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. Correctness and progress are proved in Section 6. Related work is discussed in Section 7. Section 8 presents performance experiments which demonstrate that PHyTM is highly efficient. Finally, we conclude in Section 9.

2. MODEL

We consider an asynchronous shared memory system with n 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. Although main memory is logically organized into pages, and physically organized in terms of rank, bank, device, row and column, without loss of generality, we consider the *cache line* granularity in main memory as the smallest unit of data. 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 processors 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. We do not consider any other types of failures, such process crashes, or byzantine failures. DRAM is volatile, and so are all levels of the cache hierarchy and all registers. After a power failure, only NVM still contains information.

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

If two concurrent transactions both write to the same memory location, or one reads and then the other subsequently writes, then we say that a *data conflict* has occurred. The HTM system will abort at least one of the transactions involved in each data conflict. Transactions also abort for many other reasons, e.g., if they invoke a system call, or experience an interrupt, a page fault, or an internal buffer

overflow. In particular, transactions have limitations on the number of memory addresses they can access, and exceeding these limitations will cause a *capacity abort*.

Adding support for NVM. 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 (which can cause any transactions that have *addr* in their read-sets to abort). We assume the *transparent flush* operation *TFLUSH* of Avni et al. [2], which has the same effect as *FLUSH*, except that it will not cause any transaction 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 only part of a committed transaction recorded. Therefore, we also assume the same extension to the HTM commit instruction as Avni et al. An HTM transaction is committed by invoking *txEnd*, 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).

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 transactions 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 process 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 process 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 process). 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.

4. PHYTM ALGORITHM

In PHyTM, transactions can execute on three paths: Fast HTM, Slow HTM and STM. We begin by describing STM.

4.1 The STM path

Our STM algorithm uses encounter-time locking (ETL). In ETL, transactions lock each address before accessing it. ETL makes it fairly straightforward to prove the common correctness condition *opacity* [9, 10], which intuitively states that processes cannot observe the partial results of transactions. Since a process can read an address only after locking it, and the process 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.

We can avoid deadlock by using a *TryLock* primitive instead of a *Lock* primitive. Unlike 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 processes 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 enters its write-back phase. In its write-back phase, a transaction flushes its log to NVM, performs all of its writes, and then flushes its writes 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 process, *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 process.)

4.2 The Slow HTM and Fast HTM paths

Like the STM path, Slow HTM 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, Slow HTM differs from the STM path in two ways. First, Slow HTM actually performs its writes immediately after logging them (without waiting for the log to be replayed). This only works on the HTM paths because these writes remain in the process's private cache until the transaction commits. Second, Slow HTM 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 process at some later point before the transaction commits, then the transaction will abort.

At a high level, Slow HTM *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 txEnd to atomically commit the transaction and mark its log as *completed*, so that a recovery process will replay it, should a power failure occur. Finally, Slow HTM 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 Slow HTM, it moves to the STM path.

Fast HTM is the same as Slow HTM, except that it does not need to subscribe to locks for the addresses it reads to

guarantee that it sees a consistent view of memory (i.e., that it does not see a state that would be impossible in a sequential execution). To see why, we make two observations. First, whenever a transaction on Slow HTM or Fast HTM would cause another transaction T on Fast HTM to see inconsistent state, the HTM system will cause T to abort (since both transactions are running in HTM). Second, any transaction T on Fast HTM will abort if it runs concurrently with an STM transaction in its write-back phase. This is because T begins by checking if the counter that contains the number of STM transactions currently in their write-back phases is zero (and aborts, otherwise). If this counter changes from zero to a non-zero value during T, then the HTM system will abort T because of a data conflict. Thus, reads on Fast HTM are uninstrumented.

4.3 Executing on different paths

In this section, we describe how transactions move between paths, and when transactions on different paths can run concurrently (see Figure 1).

Each transaction begins on Fast HTM. A transactions T on Fast HTM cannot run while there is an STM transaction in its write-back phase (or else T may see only some of the writes performed by the STM transaction). When a transaction on Fast HTM aborts, we record whether it aborted because there was an STM transaction in its write-back phase, or for some other reason. If a transaction T on Fast HTM aborts sufficiently many times because an STM transaction was in its write-back phase, then T moves to the Slow HTM path, where it can run concurrently with any transaction. Otherwise, if T aborts sufficiently many times for *other* reasons, it moves to the STM path. Similarly, if a transaction on Slow HTM aborts sufficiently many times, it moves to the STM path.

The STM path uses a global reader/writer-lock to guarantee progress. Each transaction on the STM 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* (which allows concurrency with other transactions on the STM path). Otherwise, the transaction acquires the global lock as a *writer*, which blocks all other transactions on the STM path. This makes it impossible for livelock to occur between transactions on the STM path, because, eventually, each transaction will exhaust its budget and resort to taking the global lock as a writer, which prevents other STM transactions from interfering with it. Although a transaction that has acquired the global lock as a writer does not run concurrently with any other transaction on the STM path, it still acquires all of its per-address locks. This allows transactions on Slow HTM to continue running concurrently with an STM transaction that holds the global lock as a writer. However, since transactions on Slow HTM acquire per-address locks, they may hold locks that are needed by this STM transaction. This makes the progress argument somewhat subtle.

5. PHYTM IMPLEMENTATION

In this section, we give the full details of the PHyTM implementation. Fast HTM, Slow HTM and the STM 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 the appropriate path) using the functions we provide. We begin by describing the underlying data structures.

Path	Fast HTM	Slow HTM	STM-R	STM-W
Fast HTM	Yes	Yes	Yes*	Yes*
Slow HTM	Yes	Yes	Yes	Yes
STM-R	Yes*	Yes	Yes	No
STM-W	Yes*	Yes	No	No

Figure 1: Table showing which paths can run concurrently. STM-R (STM-W) represents a transaction on the STM path holding the global lock as a reader (writer). *Fast HTM can run concurrently with STM as long as no STM is in its write-back phase.

```
1  type log_entry
2    int wsize        // size of the write-set
3    word* wset[]     // addresses in the write-set
4    word wdata[]     // data to be written by the txn
5    bool logged      // true if the log entry is complete
6
7  // process-local read-set data
8  process_local int rsize    // size of the read-set
9  process_local word* rset[] // addresses in the read-set
10
11 // shared data
12 shared log_entry entries[]
13 shared rwlock_t locks[]
14 shared rwlock_t globallock
15 shared int numSTMWriteback = 0
```

Figure 2: Data structures for PHyTM

5.1 Data structures

The data structures for the PHyTM implementation appear in Figure 2. Broadly, they consist of per-process log entries, process-local read sets, locks to protect memory addresses, the global reader/writer-lock introduced in Section 4, and a counter. The per-process log entries are stored in an array, *entries*, which has one entry per process. Each process has a local read set *rset* of size *rsize*. 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 counter, *numSTMWriteback* contains the number of STM transactions currently in their write-back phases.

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 processes 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* [7].

5.1.2 Per-process read-sets and write-log entries

Each process has a write-log entry (in the *entries* array) containing four variables: *wsize*, *wset*, *wdata* and *logged*. *wsize* contains the number of addresses in the write-set. *wset*

```
void STMBegin(log_entry* rec)
  if budget of transaction attempts is exhausted
    WriteLock(globallock)
  else
    ReadLock(globallock)

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

  // remember addr so we can unlock it later
  rset[rsize++] = addr
  return val

// precondition: STMRead(addr, rec) has previously been
      invoked in this transaction
void STMWrite(word* addr, word val, log_entry* 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 STMFinalize(log_entry* rec)
  // flush the log entry
  FlushLogEntry(rec)

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

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

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

Figure 3: Operations for the STM path

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 process performing the transaction. This bit indicates to the recovery process that this log entry should be replayed, should a power failure occur.

Each process also has a local read-set represented by two variables: *rsize* and *rset*, which are analogous to *wsize* and *wset*. Unlike the write-set, the read-set is not (explicitly) flushed to NVM, and is never used by the recovery process. The read-set is only used by transactions on the STM path, which use it simply to keep track of which addresses they have locked as readers (so they can be unlocked after the transaction is committed).

5.2 The STM path

The STM path provides four operations: *STMBegin*, which starts a transaction, *STMRead*, which replaces a standard read from memory, *STMWrite*, which replaces a standard write, and *STMFinalize*, which commits a transaction. The pseudocode for the STM path operations appears in Figure 3.

An *STMBegin* operation simply acquires the global lock as a reader (if the transaction's budget for attempts has not yet been exhausted) or a writer (if it has).

An *STMRead* operation invokes *TryReadLock* to acquire a read-lock, reads the address, and saves it in its read-set. An *STMWrite* operation invokes *TryWriteLock* to acquire 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. (If the process executing *TryWriteLock* currently holds the lock as a reader, and there are no other readers, then *TryWriteLock* upgrades the read-lock to a write-lock.) The *STMWrite* 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[1]). If *STMRead* or *STMWrite* 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 process invokes *STMFinalize*. *STMFinalize* flushes the write-log entry to NVM, and then indicates that the transaction has entered its write-back phase by invoking *FetchAndIncrement* on *numSTMWriteback*. It then sets and flushes a logged bit in the log entry, which indicates that it is ready to be replayed by the recovery process if a power failure occurs. The transaction is *committed* precisely when the logged bit reaches NVM. Next, *STMFinalize* invokes a function called *ReplayLogEntry* (which appears in Figure 6) to replay the transaction's 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. After all of the transaction's writes are performed and flushed, *STMFinalize* performs fetch and decrement on *numSTMWriteback* (which indicates that the transaction is no longer in its write-back phase). Finally, *STMFinalize* unlocks all of its locks and prepares its log entry for reuse by the process's next transaction.

5.3 The Slow HTM path

Slow HTM provides the same operations as the STM path, but their names have a "SlowHTM" prefix instead of "STM." Pseudocode for these operations appears in Figure 4.

A *SlowHTMBegin* operation starts by determining whether the transaction should move to the STM path (because it has exhausted its budget of attempts). If so, the transaction moves to the STM path. Otherwise, *SlowHTMBegin* simply starts a hardware transaction.

A *SlowHTMRead* operation reads the lock state for the address being read, and aborts if the lock is held by another process. If the lock is not held, then *SlowHTMRead* reads and returns the address. A *SlowHTMWrite* operation tries to lock the address being written as a writer, and aborts if it fails to do so. This lock grants exclusive access to the address being written, and exclusive permission to store that address in its write-log entry. If *SlowHTMWrite* successfully acquires the lock, then it adds the address and the value to be written to the transaction's write-log entry. Finally, it performs its actual write.

[1]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 process does not explicitly flush its modifications to its log entry, yet.

```
64 void SlowHTMBegin(log_entry* rec)
65   if budget of transaction attempts is exhausted
66     move to STM path
67   else
68     invoke txBegin
69
70 word SlowHTMRead(word* addr, log_entry* rec)
71   // check if addr is write-locked
72   rwlock_t* lock = GetLockAddr(addr)
73   if IsWriteLocked(lock) then abort and retry
74   return *addr
75
76 void SlowHTMWrite(word* addr, word val, log_entry* rec)
77   // try to write-lock addr
78   rwlock_t* lock = GetLockAddr(addr)
79   if !TryWriteLock(lock) then abort and retry
80
81   // add <addr, val> to the write-log
82   int wsize = rec->wsize
83   rec->wset[wsize] = addr
84   rec->wdata[wsize] = val
85   rec->wsize++
86   // perform the write
87   *addr = val
88
89 void SlowHTMFinalize(log_entry* rec)
90   // flush the log entry
91   FlushLogEntry(rec)
92   // atomically commit (to cache) and
93   // simultaneously set rec->logged in NVM
94   // to indicate the log entry is ready to be replayed
95   txEnd(rec->logged)
96
97   // replay the log entry to flush all writes
98   ReplayLogEntry(rec, false)
99   Unlock all locks
100
101  ResetLogEntry(rec)
102  rec->attempts = 0
```

Figure 4: Operations for Slow HTM

To commit a transaction on Slow HTM, a process invokes *SlowHTMFinalize*. *SlowHTMFinalize* flushes the write-log entry to NVM, and then invokes txEnd. This 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 process if a power failure occurs). After this, *SlowHTMFinalize* invokes *ReplayLogEntry* (see Figure 6) 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 *STMFinalize*, this invocation of *ReplayLogEntry* does not need to perform the transaction's writes, since they are already performed as part of the hardware transaction.) Finally, *SlowHTMFinalize* unlocks all of its locks and prepares its log entry for reuse by the process's next transaction.

5.4 The Fast HTM path

Fast HTM provides the same operations as the Slow HTM path, but their names have a "FastHTM" prefix instead of "SlowHTM." Pseudocode appears in Figure 5.

FastHTMWrite and *FastHTMFinalize* are identical to their Slow HTM counterparts. *FastHTMBegin* first checks if the budget for transaction attempts is exhausted, and, if so, the transaction moves to Slow HTM or the STM path, as appropriate. Otherwise, *FastHTMBegin* invokes *txBegin* to start a hardware transaction, and then checks if *numSTMWriteback* is greater than zero. If so, the transaction aborts. *FastHTMRead* is uninstrumented: it simply reads the address and returns.

```
103 void FastHTMBegin(log_entry* rec)
104   if budget of transaction attempts is exhausted
105     move to STM path or Slow HTM path as appropriate
106   else
107     invoke txBegin
108     if numSTMWriteback > 0 then abort
109
110 word FastHTMRead(word* addr, log_entry* rec)
111   return *addr
112
113 void FastHTMWrite(word* addr, word val, log_entry* rec)
114   // identical to SlowHTMWrite
115
116 void FastHTMFinalize(log_entry* rec)
117   // identical to SlowHTMFinalize
```

Figure 5: Operations for Fast HTM

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

Figure 6: Functions common to all paths

5.5 Recovery

After a power failure, the recovery process runs a simple procedure called *Recovery* (see Figure 6). 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 processes can resume normal operation. So, *Recovery* begins by unlocking all processes' locks. (The recovery process 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 *STMFinalize*, *SlowHTMFinalize* and *FastHTMFinalize* to complete a transaction once a log entry is flushed.

ReplayLogEntry first checks if the log entry has its *logged* bit set. If so, the transaction has been committed, and its log must have been flushed to NVM. Next, the transaction's writes are performed at line 142 (because $doWrites = true$ when *ReplayLogEntry* is invoked by *Recovery*). (Note that the recovery process performs these (apparently redundant) writes even for hardware transactions, despite the fact that *SlowHTMFinalize* and *FastHTMFinalize* invoke *ReplayLo-*

gEntry with *doWrites* = $false$, and do not perform these writes. Here, these writes are necessary, because after a hardware transaction commits, but before its writes are flushed to NVM, they may be lost to a power failure.) We briefly argue that, when the power failure occurred, the process performing the transaction held write-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 is reset to zero at the end of *ReplayLogEntry*. It follows that a power failure occurred before the process that was running this transaction could finish its invocation of ReplayLogEntry in *STMFinalize* (at line 57), *SlowHTMFinalize* (at line 98), or *FastHTMFinalize*. In each case, the process still held write-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.

5.6 Optimizing with non-transactional reads

In this section, we describe an optimization to PHyTM that can be applied in HTM system that allows processes to perform non-transactional reads and writes while inside a transaction. Note that the *SlowHTMRead* function reads both the value stored at an address and the state of its lock. A paper by Riegel et al. [19] 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 protected by the lock. In PHyTM, this optimization is quite natural, since locks are already taken by the HTM paths for logging, and not simply because of their interactions with the STM path. PHyTM can also use non-transactional reads and writes to maintain the per-process write-log entries (since each write-log entry is accessed only by the single process that writes to it, and by the recovery process, which runs alone in the system). These optimizations would reduce the number of locations to which transactions subscribe, which reduces the likelihood of capacity aborts.

6. CORRECTNESS

In this section, we prove that the PHyTM algorithm provides linearizable transactions, and that the algorithm is deadlock- and livelock-free.

6.1 Progress

It is fairly straightforward to make an informal argument 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, transactions will eventually exhaust their budgets for transactional attempts on all paths, and will each attempt to take the global lock as writers, at which point one of the transactions T will acquire the global lock as a writer and commit a transaction (yielding a contradiction). However, the full progress proof is deceptively subtle.

Definition 1. *We refer to operations starting with "FastHTM" as* ***Fast HTM operations****, operations starting with "SlowHTM" as* ***Slow HTM operations****, and operations starting with "STM" as* ***STM 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 STMBegin.*

Proof. Recall that we assumed *TryReadLock* and *TryWriteLock* return immediately if the lock is held. Because of this assumption, *STMRead*, *STMWrite*, *SlowHTMRead*, *SlowHTMWrite*, *SlowHTMBegin*, *FastHTMRead*, *FastHTMWrite* and *FastHTMBegin* are straightline code. *STMFinalize*, *SlowHTMFinalize* and *FastHTMFinalize* are straightline code, apart from their invocations of *FlushLogEntry* and *ReplayLogEntry*. *FlushLogEntry* and *ReplayLogEntry* are straightline code except for their loops, which 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 128 in *ResetLogEntry*, where it is set to zero, and in *FastHTMWrite*, *SlowHTMWrite* and *STMWrite*, 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 processes take steps infinitely often, then transactions commit infinitely often.)*

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

Claim 1. Eventually, every process invokes only STM operations.

Suppose not, to obtain a contradiction. Then, some process p executes Fast HTM or Slow HTM operations infinitely often. Since no transactions can commit after time t, p must perform infinitely many Fast HTM or Slow HTM operations without committing. Since transactions have finite read- and write-sets, p's must abort infinitely often, which means it must eventually exhaust its budget for attempts on the HTM paths, and move to the STM path, which is a contradiction.

Claim 2. Eventually, every process either invokes *STMBegin* infinitely often, or spins forever at line 18 or line 20 in *STMBegin*.

Suppose not, to obtain a contradiction. Then, some process p invokes *STMBegin* finitely many times, and does not spin forever in *STMBegin* (the only place where unbounded spinning could occur). 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 processes that invoke *STMBegin* infinitely often. Since the global lock used by PHyTM is deadlock-free, it is impossible for *every* process to spin forever in *STMBegin*. Thus, Claim 2 implies that σ is non-empty. By a similar argument to the proof of Claim 1, every process in σ eventually exhausts its budget of transactional attempts on the STM path. Therefore, eventually, every invocation of *STMBegin* by a process in σ attempts to acquire the global lock as a writer at line 18. Since the global lock is deadlock-free, eventually some process 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

We now show that PHyTM transactions are linearizable. In this section, we assume no power failures occur. (We consider power failures in the next section.)

DEFINITION 2. *A **transaction attempt** by a process p is any interval starting with an FastHTMBegin (resp. SlowHTMBegin or STMBegin) by p and ending with the next FastHTMFinalize (resp. SlowHTMFinalize or STMFinalize) 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 attempts by one process.

DEFINITION 3. *A transaction attempt on Fast HTM **commits** at its execution of txEnd. A transaction attempt on Slow HTM **commits** at its execution of txEnd (at line 95). A transaction attempt on the STM path **commits** at its execution of TFLUSH (at line 54).*

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

Linearization points

- Each committed transaction is linearized precisely at the moment it commits (see Definition 3).
- Each aborted transaction attempt on Fast HTM is linearized at the last time it accesses a lock in *FastHTMWrite*.
- Each aborted transaction attempt on Slow HTM is linearized at the last time it accesses a lock in *SlowHTMRead* (line 73) or *SlowHTMWrite* (line 79).
- Each aborted transaction attempt on the STM path is linearized at the last time it accesses a lock in *STMRead* (line 24) or *STMWrite* (line 37).

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 in STMWrite (lines 43-45), SlowHTMWrite (lines 83-85) or FastHTMWrite.*
3. *T locks addr as a writer (in STMWrite at line 37, in SlowHTMWrite at line 79, or in FastHTMWrite), before adding addr to its write-log entry, before writing to addr, before committing and before flushing addr to NVM. T continuously holds this lock until after it writes to addr, after it commits and after it flushes addr to NVM.*

PROOF. Suppose T executes on the STM path. Then T must write to *addr* at line 142 of *ReplayLogEntry*. Prior to invoking *ReplayLogEntry* at line 57, it commits at line 54 (Claim 1). Claims 2 and 3 are immediate from the code.

Now, suppose T executes on Slow HTM. Since *addr* is not a lock or a part of a log entry, T writes to it in *SlowHTM Write* at line 87 (inside a hardware transaction). Therefore, T must commit in order to change *addr* in main memory (Claim 1). Claim 2 is immediate from the code. We now prove Claim 3. Before T commits, it writes to *addr*. Just before T writes to *addr*, it tries to lock *addr* as a writer in *SlowHTMWrite* at line 79. Since T commits, it must successfully lock *addr*. From the code, T continuously holds this lock until it releases all locks in *SlowHTMFinalize* at line 99, which is after T flushes addr to NVM in its invocation of *ReplayLogEntry* (at line 98 of *SlowHTMFinalize*), which is after T commits in *SlowHTMFinalize* at line 95. □

LEMMA 4. *Let T be a transaction attempt with addr in its write-set, t_c be when T commits, and t_u be when T releases its write-lock on addr. If T commits, then addr continuously contains the last value v written by T starting from some time t_v ($t_c \leq t_v \leq t_u$) until a write-lock is next acquired on addr after t_u by a committed transaction attempt.*

PROOF. Let t_w be when T writes to *addr* and t_f be when T flushes *addr* to NVM. By Lemma 3, T continuously holds a write-lock on *addr* from before $\min\{t_w, t_f, t_c\}$ until after $\max\{t_w, t_f, t_c\}$. Furthermore, no other transaction can change *addr* while T holds a write-lock. It follows that *addr* contains v immediately after t_f, which is before t_u. This value is not changed again by T, and cannot be changed by another committed transaction attempt until a write-lock is next acquired on *addr* after t_u (by Lemma 3). □

LEMMA 5. *Let R be an invocation by a transaction attempt T of STMRead on an address addr. If T commits, then R returns the value v written by the last transaction T' with addr in its read-set that commits before T commits.*

PROOF. Let t_r be when R executes line 28. Our goal is to prove that *addr* contains v at t_r.

Claim: *addr* contains v at some time after T' commits and before t_r. By Lemma 4, *addr* contains v at some time t_v after T' commits, and before it releases its locks. Just before t_r, T must execute line 24 in *STMRead*, where it sees that *addr* is not locked by a writer. Thus, T' must release its write-lock on *addr* before t_r, and, hence, t_v is before t_r.

Claim: *addr* does not change between t_v and t_r.

Suppose, to obtain a contradiction, that *addr* changes between t_v and t_r. By Lemma 4, *addr* does not change until a committed transaction attempt acquires a write-lock on *addr* after t_v. Thus, a committed transaction T'' must acquire a write-lock on *addr* between t_v and t_r. By Lemma 3, T'' must commit before T acquires its write-lock on *addr*, which implies that T'' is committed between T' and T. However, we assumed that T' is the last transaction with *addr* in its write-set that commits before T commits. □

LEMMA 6. *Let R be an invocation by a transaction attempt T of SlowHTMRead on an address addr. If T commits, then R returns the value v written by the last transaction T' with addr in its read-set that commits before T commits.*

PROOF. Although *SlowHTMRead* does not explicitly acquire read-locks, it subscribes to the state of the lock for each address it reads, and sees that the lock is not held by a writer. (Moreover, since reading the lock state causes the HTM system to subscribes to it, T will abort if any concurrent transaction attempt acquires the lock.) Thus, the value it reads at line 74 is identical to value it would read if it had explicitly acquired a read-lock on the address. Consequently, the proof is the same as the proof of Lemma 5. □

LEMMA 7. *Let R be an invocation by a transaction attempt T of FastHTMRead on an address addr. If T commits, then R returns the value v written by the last transaction T' with addr in its read-set that commits before T commits.*

PROOF. Let t_r be when R executes line 28. Our goal is to prove that *addr* contains v at t_r.

Claim: *addr* contains v at some time after T' commits and before t_r. By Lemma 4, *addr* contains v at some time t_v after T' commits, and before it releases its locks. It follows

that t_v is before T commits. Suppose t_v is after t_r to obtain a contradiction. Then *addr* is changed after t_r and before T commits. Therefore, the HTM system will abort T due to a data conflict. However, we assumed that T commits.

Claim: *addr* does not change between t_v and t_r. Suppose, to obtain a contradiction, that *addr* changes between t_v and t_r. By Lemma 4, *addr* does not change until a committed transaction attempt acquires a write-lock on *addr* after t_v. Thus, a committed transaction T'' must acquire a write-lock on *addr* between t_v and t_r. By Lemma 3, T'' must commit before T acquires its write-lock on *addr*, which implies that T'' is committed between T' and T. However, this is a contradiction, since we assumed that T' is the last transaction with *addr* in its write-set that commits before T commits. □

6.3 Recovery

In this section, we prove the correctness of the *Recovery* procedure that is invoked by the recovery process 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.*

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

LEMMA 8. *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 write-locks on all addresses in its write-set. □

LEMMA 9. *Every transaction attempt that commits before a power failure either terminates (meaning its invocation of FastHTMFinalize, SlowHTMFinalize or STMFinalize 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 10. *Immediately after the recovery process finishes executing the Recovery procedure, the contents of shared memory are exactly what they would be if all transaction attempts that had committed 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 8 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 [15]. There are already working prototypes of NVM such as phase-change memory (PCM) [15], spin-torque-transfer RAM (STT-RAM) [12], and memristors [20], 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 [21] 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 [21], which was published at the same time as NV-Heaps, is word-based, and is derived from TinySTM [8]. 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-process 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 [22, 16] use HTM for synchronization. However, these databases still use flush data to disk to achieve persistence.

In [23] they design new hardware to track dependencies among transactions, and use it to decide when to flush transactional data to NVRAM to create a persistent HTM. Their design has a centralized scoreboard, which is not scalable. While [23] involves nontrivial hardware additions (with potential scalability issues), PHTM [2] is a persistent version of HTM, for machines that provide NVM, which implies only changes in HTM microcode. It writes a persistent bit to NVM as part of HTM commit execution, and uses software to log the write-set in a private NVM buffer for safety. This way if a power or hardware failure ever occurs, the contents of shared memory can be recovered. Additionally, PHTM has uninstrumented reads, which can be executed at hardware speed.

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 [11], but the first algorithms that allow a fast path concurrency with the slow path were introduced in 2006 [6, 14]. 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 [5]. (In this direction, HTM is also used in the commit phase of an STM transaction, which eliminates the need for HTM to subscribe to locks [17].)
- Attaching a versioned lock or a traditional lock to each address, which is read by each HTM read operation, and is acquired by both paths for each write, in the HyLSA algorithm [19]. This algorithm greatly increases the size of HTM transactions, because locks must be read by each HTM read operation. 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. EXPERIMENTAL ANALYSIS

In this section we study the performance of PHyTM using various workloads. We use an Intel Core i7-4770 3.4 GHz Haswell processor with 4 cores, each with 2 hyperthreads. Each core has private L1 and L2 caches, whose sizes are 32 KB and 256 KB respectively. There is also an 8 MB L3 cache shared by all cores. We use the HTM provided by the hardware and emulate NVM.

Emulation. Our PHyTM implementation is based on the NVM emulation scheme used by PHTM [2]. In our experiments, the atomic assignment and flush of the *logged* bit by the commit instruction is emulated by avoiding any simulated power failures between the HTM commit and when the bit is set. Since writes in NVM are expected to be slower than writes to DRAM, we inserted delays to simulate writes to NVM.

Workloads. Following [24], we implemented the Yahoo! Cloud Serving Benchmark (YCSB). This is a collection of workloads that are representative of large-scale services created by Internet-based companies [4]. For all of the YCSB experiments in this paper, we used a single table with 20 million records. Each YCSB tuple has a single primary key column. The DBMS creates a single hash index for the primary key. Each transaction in the YCSB workload accesses

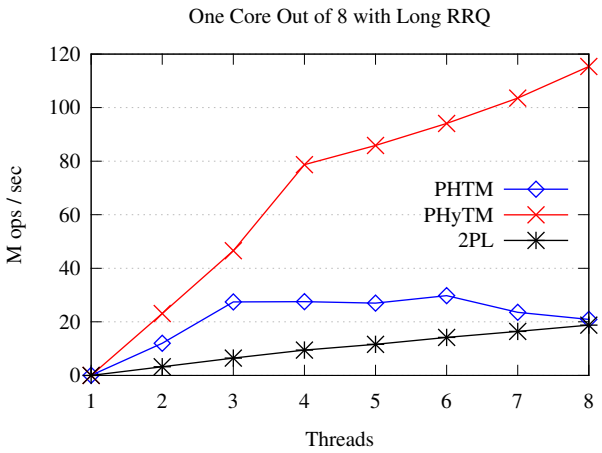

Figure 8: YCSB workload where one process performing large reading range queries

a predefined number of records, where 16 is the default. In some of our workloads, we also perform large transactions that access 256 records. Since these large transactions access so many records, they are likely to cause capacity aborts. If the accesses in a transaction are writes, we call the transaction a *writing range query* (WRQ). If they are reads, then we call the transaction a *reading range query* (RRQ). These transactions simply read and write the contents of records, and do not perform any additional computation. All transactions are independent. That is, the behaviour of a transaction does not depend on the result of a previous transaction.

Algorithms. We implemented the YCSB benchmark using PHyTM, PHTM, and two-phase locking (2PL), which uses fine-grained locking on the rows a table. 2PL was recently experimentally studied on simulated systems with more than one thousand processors by Yu et al. [24], and was shown to be quite scalable. 2PL typically incurs some overhead to perform deadlock detection. However, our workloads contain no deadlocks, and our implementation of 2PL takes advantage of this by eliminating deadlock detection.

Results. The results of our experiments appear in Figure 7 and Figure 8. Broadly speaking, when there are few aborts, PHyTM behaves similarly to PHTM. This is because, as long as there is no transaction on the STM path, both hybrid TMs have no instrumentation overhead for reads, and the overhead of writing to NVM dominates any performance differences in write-heavy workloads. The performance of 2PL in write-heavy workloads is also dominated by the overhead of writing to NVM. The NVM overhead for writes is the same for all algorithms in our benchmark, so they all exhibit approximately the same performance in write-only scenarios, as Figure 7c demonstrates. However, Figure 7b shows that PHTM and PHyTM read operations are an order of magnitude faster than 2PL.

Figure 7d shows a YCSB workload that performs half RRQs and half WRQs. In this workload, the overhead of writing to NVM is more significant than any algorithm-dependent performance difference, so the performance of 2PL is fairly close to that of PHTM and PHyTM.

The most significant shortcoming of HTM is that the size of a transaction is limited by the capacity of the cache. Hardware transactions that access sufficiently many locations are doomed to abort, and the probability of aborting increases

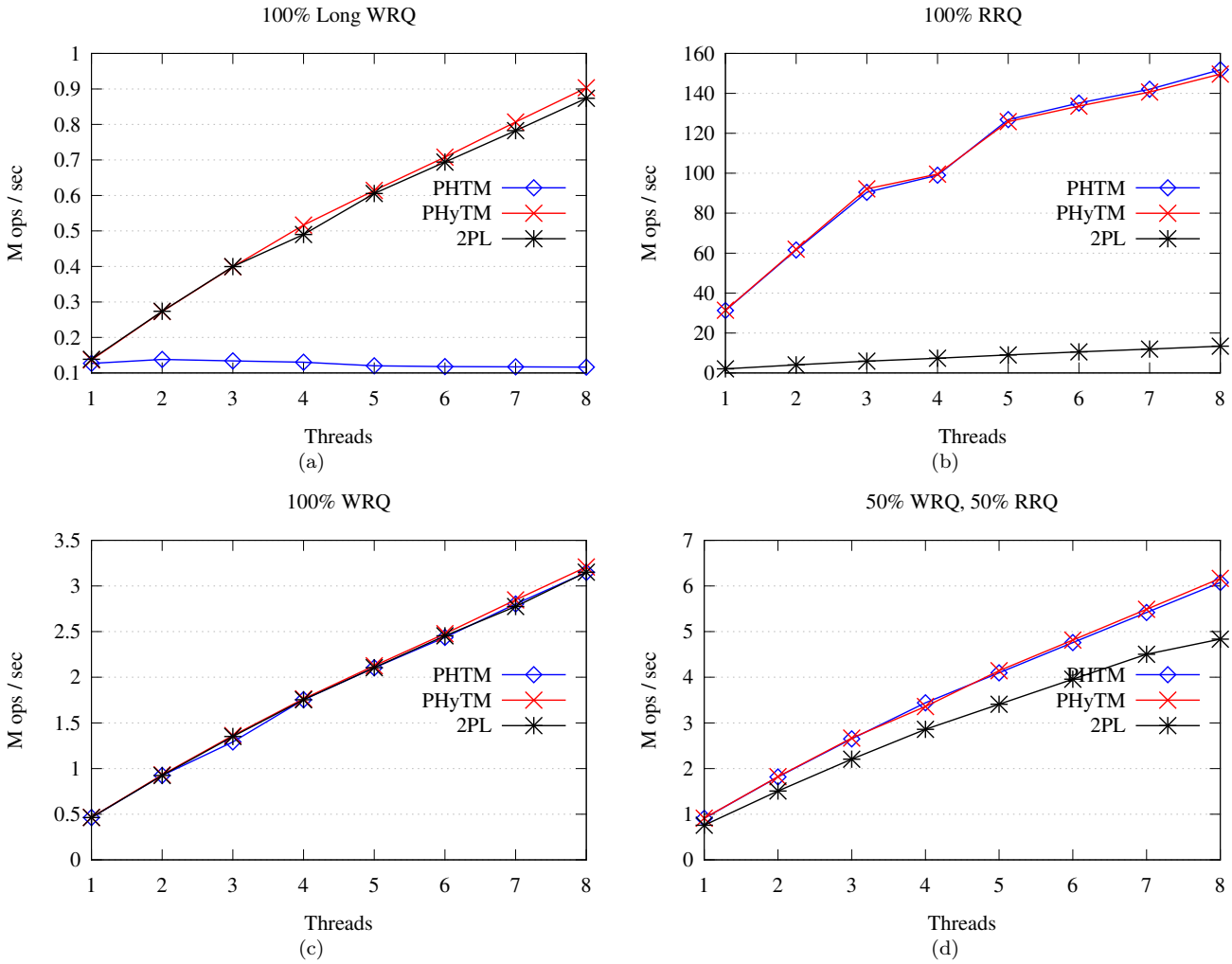

Figure 7: YCSB workloads for different transaction mixes

as the size of a transaction grows. PHTM includes an STM fallback path to allow such transactions to succeed, but the STM path acquires a global lock, so transactions on the STM path are serialized. This represents a severe bottleneck, especially as systems with HTM support become increasingly parallel (with configurations supporting hundreds of threads currently possible).

To study what happens when a nontrivial fraction of transactions fail in hardware, and must be executed in software, we added a workload containing large transactions that are unlikely to commit in hardware. When all transactions are large WRQs, we see in Figure 7a that PHTM is not scalable at all, while PHyTM is as scalable as 2PL. PHyTM scales because STM transactions can run concurrently with each other, and with other hardware transactions. The overhead that PHyTM incurs by optimistically trying transactions in hardware before falling back to software does not prevent it from matching the performance of 2PL (which never has to abort), even in this workload with many aborts. We believe this is because PHyTM avoids performing expensive writes/flushes to NVM until after it commits. Thus, aborted transactions avoid this overhead.

In the workload consisting entirely of large WRQs, PHyTM and 2PL achieve approximately the same throughput, and perform an order of magnitude better than PHTM. In the workload consisting entirely of RRQs, PHTM and PHyTM achieve approximately the same throughput, and perform an order of magnitude better than 2PL.

The most significant advantage of PHyTM over PHTM and 2PL becomes clear when a subset of the processes are running transactions on the STM path while most of the processes successfully commit transactions in HTM. This situation is demonstrated in Figure 8, where only one process is executing large RRQs (which are unlikely to succeed in hardware, and often run on the STM path), and the other processes execute smaller transactions that are typically able to commit in hardware.

When the smaller transactions are mostly read-only, as in Figure 8, PHyTM is an order of magnitude faster than its competitors. The STM path of PHTM performs reads with no overhead, so it is much faster than 2PL (which must acquire locks) at low process counts. However, since the STM path of PHTM acquires a global lock, it does not scale 2PL, which scales in this workload, ties PHTM with eight concurrent processes, but suffers from the high cost of acquiring locks. Whenever there is no transaction on the STM path in its write-back phase, PHyTM transactions can run on Fast HTM, where their read operations have no overhead. Furthermore, even when a transactions on the STM path is in its write-back phase, PHyTM transactions

can run on Slow HTM, where their read operations can simply read the state of locks instead of acquiring them.

9. 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 was 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.

10. REFERENCES

[1] Intel architecture instruction set extensions programming reference.

[2] H. Avni, E. Levy, and A. Mendelson. Hardware transactions in nonvolatile memory. In *Proc. of 29th Int. Sym., 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] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[5] 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 *Proc. of 16th Int. Conf. on Arch. Supp. for Prog. Lang. and Oper. Sys., ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 39–52, 2011.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. 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.

[7] 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.

[8] 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 Sym. on Principles and Practice of Par. Prog.*, pages 237–246. ACM, 2008.

[9] 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.

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

[11] 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.

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

[13] Intel. Private communication with hardware engineers, February 2016.

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

[15] 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.

[16] 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.

[17] 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.

[18] D. Narayanan and O. Hodson. Whole-system persistence. *SIGPLAN Not.*, 47(4):401–410, Mar. 2012.

[19] 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.

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

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

[22] 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.

[23] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen. Persistent transactional memory. *Computer Architecture Letters*, 14(1):58–61, 2015.

[24] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.