PHyTM: Persistent Hybrid Transactional Memory

Hillel Avni
Huawei Technologies
European Research Institute
hillel.avni@huawei.com

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

ABSTRACT

Processors with hardware support for transactional memory (HTM) are rapidly becoming commonplace, and processor manufacturers are currently working on implementing support for upcoming non-volatile memory (NVM) technologies. The combination of HTM and NVM promises to be 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 full benefit of hardware transactional memory. In this paper, we propose a *persistent hybrid TM* algorithm called PHyTM for systems that support NVM and HTM. PHyTM allows hardware assisted ACID transactions to execute concurrently with pure software transactions, which allows applications to gain the benefit of persistent HTM while simultaneously accommodating unbounded transactions (with a high degree of concurrency). Experimental simulations 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, but the following model is currently leading. Systems can contain only NVM, or a combination of DRAM and NVM. Data is asynchronously flushed to NVM at any time, and without the programmer's knowledge. A programmer can also explicitly cause data 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 data has been flushed to NVM.

Proceedings of the VLDB Endowment, Vol. 10, No. 4

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 [20]. This approach could allow applications to avoid any runtime overhead associated with providing persistence (since the entire processor state would be persistent). 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 [14]. Operating under this assumption, Intel is currently designing new and efficient flush instructions (CLWB and CLFLUSHOPT) with NVM in mind [13]. 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.) HTM allows programmers to execute arbitrary blocks of code in transactions, which either commit and appear to occur atomically, or abort and have no effect on the contents of memory. 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 one can imagine 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 HTM implementations and NVM proposals is particularly interesting, because transactions must appear to be atomic, but writes performed by the fallback path can trickle asynchronously to NVM 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 NVM, when a power failure could cause some or all of 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. It appears that existing HTM implementations will have to be modified to support NVM, since at least one bit should be communicated to NVM atomically, as part of an HTM commit, to indicate that the transaction has committed (otherwise, power failures might cause committed transactions to be lost). Avni et al. 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 capability is used by PHTM to maintain *redo logs* that ensure committed transactions are not lost. Specifically, it 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 executes all transactions sequentially, 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. Like PHTM, PHyTM uses redo logging to facilitate recovery after power failures. PHyTM provides atomic 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 in rare situations, transactions on the STM path may take a coarse-grained lock that 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 data 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, it moves to the STM path to guarantee progress.

In-memory databases (IMDBs) such as HANA (SAP), Hekaton (Microsoft), TimesTen (Oracle), MemSQL and VoltDB are frequently used in single-node configurations for data analytics, and applications where response time is critical, such as telecommunications networks and real-time financial services. In their implementations, IMDBs use a small set of well-known synchronization mechanisms to coordinate access by processes to internal data structures. Common synchronization mechanisms include 2-phase locking (2PL), optimistic concurrency control (OCC) and multiversion concurrency control (MVCC). Recent work has shown that SAP HANA can obtain significant performance benefits from current HTM implementations [15], and improved fault tolerance from proposed NVM implementations [22]. One promising application of our work is, thus, to use PHyTM as a new synchronization mechanism for IMDBs, harnessing the power of HTM to reduce synchronization overhead over the traditional approaches while simultaneously adding support for NVM.

To demonstrate the potential of such an approach, we performed experiments on an Intel system with HTM support and simulated NVM support. Using the Yahoo! Cloud Serving Benchmark (YCSB) and TPC-C benchmark, we compared the performance of a simple IMDB implemented with four different synchronization mechanisms: 2PL, OCC, PHTM and PHyTM. The results show that PHyTM is highly efficient and scalable, and often significantly outperforms the other approaches.

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. Correctness and progress are proved in Section 5. Related work is discussed in Section 6. Section 7 presents performance experiments. Finally, we conclude in Section 8.

2. MODEL

We consider an n process asynchronous shared memory system with support for HTM and NVM.

2.1 Memory

The memory is organized into a hierarchy where the lowest level, *main memory*, consists entirely of NVM. (If the system also contains DRAM, then the logical memory space is typically partitioned into persistent and non-persistent addresses. For simplicity, we consider systems with no DRAM.) 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 lower in the memory hierarchy are orders of magnitude slower than operations higher in the hierarchy. NVM is expected to be slower than DRAM for write operations, but at least as fast for read operations.

Data trickles asynchronously from the cache levels into NVM, *at any time*, and without the programmer's knowledge. Data can also be explicitly flushed to NVM 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.

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. All levels of the cache hierarchy and all registers are volatile. 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. This 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 be dangerous, such as forcibly releasing locks that were held by other processes before the failure.

2.3 Hardware transactional memory

We consider Intel's implementation of HTM. A process p begins a transaction by invoking a procedure called *xbegin*, which either returns *OK*, to indicate that the processor is now executing in transactional mode, or an *abort reason* (which we discuss below). In transactional mode, each time p performs a read (resp., write) at an address, the address is inserted into the transaction's *read-set* (resp., *write-set*). We call the union of a transaction's read-set and write-set its *data-set*. If the data-set of one transaction intersects the write-set of another concurrent transaction, then we say these two transactions have a *data conflict*. The HTM system will abort at least one of the transactions involved in each data conflict. Process p is able to abort its own current transaction by invoking *xabort*.

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*. Capacity aborts are non-deterministic, and hardware transactions are more likely to experience capacity aborts as their data-sets grow in size. To avoid a capacity abort, a transaction's data-set must (at least): fit in the L1 cache, avoid cache associativity conflicts, and avoid evictions of cache lines loaded by the transaction. Additionally, when two hyperthreads are running on one core, the core's cache is shared between the two hyperthreads. This effectively halves the L1 cache capacity for each thread, which can cause additional capacity aborts.

Whenever a transaction T by process p is aborted, p stops taking steps in the transaction, and its control flow returns to just before it invoked *xbegin* (i.e., just before the transaction began). Consequently, the next step taken by p will be an invocation of *xbegin*. This invocation will return a *reason* for the abort (e.g., *conflict* or *capacity*). Thus, *xbegin* is typically invoked with the following pattern: "**if** *xbegin*() = *OK* **then** {transaction body} **else** {abort handler}."

Adding support for NVM. It is important to note that hardware transactions do not directly modify main memory. Instead, all transactional writes are performed *in cache*, and all affected cache lines are then flushed to main memory, one-by-one, after the transaction has committed. The cache coherency protocol ensures that the affected cache lines *appear* to be flushed atomically to main memory when the transaction commits. However, if a power failure were to occur in the middle of flushing, then the cached information needed to maintain the illusion of atomicity would be lost.

One way to avoid this problem is to have each transaction log its writes and flush the log to NVM *before* committing, so that a recovery process would have enough information to complete any transaction that committed before a power failure. Naturally, this approach requires an implementation of *FLUSH* that can be used inside transactions (without simply causing them to abort). Following Avni et al. [2], we assume a *transparent flush* (*TFLUSH*) that does not cause transactions to abort. Additionally, since the log is flushed *before* the transaction is committed, there must be a way for a recovery process inspecting a transaction's log to tell whether the transaction committed before the power failure. Thus, as in [2], we assume that the *xend* procedure is extended to take the address of a single bit as its argument. This bit is atomically set and flushed to NVM at the same time as the transaction is committed in cache. Atomically communicating *a single bit* to NVM when a transaction is committed appears to be the smallest reasonable change to existing HTM protocols.

3. LOGGING IN PHTM

Since we build on the logging mechanism used by 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 entry is no longer necessary (and will no longer be used by a recovery process). Holding locks until the log entry 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. Each path provides a set of operations for starting and committing transactions, and reading and writing memory locations. Pseudocode for these operations appears in Figure 2. These operations are not directly invoked from user code. Instead, a user simply invokes operations provided by the compiler for *starting*

```
type log_entry
  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
// process-local read-set data
process_local int rsize    // size of the read-set
process_local word* rset[] // addresses in the read-set
// shared data
log_entry entries[]
rwlock_t locks[]
rwlock_t stmLock
int numSTMWriteback = 0
```

Figure 1: Data structures for PHyTM

and *committing* transactions, and the compiler automatically instruments the user's code so that it executes transactions (on the appropriate path) using the operations we provide. We begin by describing the underlying data structures.

4.1 Data structures

The data structures for the PHyTM implementation appear in Figure 1. Broadly, they consist of: a reader/writer-lock (*stmLock*), a counter (*numSTMWriteback*), per-address locks, per-process log entries, and process-local read sets. *stmLock* and *numSTMWriteback* are described in Section 4.2. We now describe the other data structures.

Per-address locks. Each memory location conceptually has an associated lock that guards it. To allow greater concurrency between read-heavy transactions, we use reader/writer-locks, which can be acquired by one writer or multiple readers. To avoid the enormous space overhead of dedicating a unique lock to each memory address, we use a smaller number of locks, which are stored in an array called *locks*. These locks are accessed via a function, *GetLockAddr*(*addr*), which *hashes* a memory address *addr* into the array of locks (effectively guarding multiple addresses with each lock). This dramatically reduces the space complexity of PHyTM, but it can cause false conflicts if processes simultaneously try to acquire locks on two different addresses that are associated with the same lock. The same approach was taken by Dice et al. in possibly the most well known STM, *TL2* [8].

Observe that, since current HTM implementations detect conflicts at a cache line level of granularity, storing multiple locks per cache line may cause suprious conflicts. If this is a concern, then locks in the array can be padded.

Per-process write-log entries. Each process has a write-log entry (in the *entries* array) containing four variables: *wsize*, *wset*, *wdata* and *logged*. *wsize* is the number of addresses in the write-set. *wset* is an array that contains all of the addresses in the write-set. *wdata* is an array that contains the values written to these addresses. *logged* is true if the log entry is *complete*, meaning that all of its contents have been written and flushed to NVM, and false otherwise. This bit indicates to the recovery process that this log entry should be replayed if a power failure occurs.

Process-local read-sets. 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 or used by the recovery process. The read-set is used only by transactions on the STM path, which use it simply to keep track of which addresses they have locked as readers (so the addresses can be unlocked once the transaction commits or aborts).

4.2 The STM path

Our STM algorithm implements two-phase locking (2PL) with encounter-time order (ETO). Each transaction first locks all of the addresses it will access (in the order it first encounters them), then it performs any modifications to these addresses, and finally releases all locks. To avoid deadlock, we use *try-locks*, which immediately return false, instead of blocking, when a lock cannot be acquired. If a process fails to acquire a (try-)lock, it releases all of its locks and tries again. In the unlikely event that a process *repeatedly* fails to acquire the locks it needs (after many attempts), it locks the entire STM path. (As an alternative to try-locks, one could use a standard deadlock detection algorithm. However, this would likely be less efficient, except under very high contention.)

2PL and ETO make it fairly straightforward to prove the common transactional memory correctness condition *opacity* [10], which is stronger form of serializability. Whereas serializability requires *committed* transactions to be consistent with a single execution history, opacity requires *all* transactions to be consistent with a single execution history (even transactions that will eventually abort). Intuitively, it guarantees that processes cannot observe *partial effects* of transactions. (I.e., a process cannot see *any* of a transactions writes until *all* have been performed.) Consider a transaction T that writes to several addresses. 2PL and ETO require T to lock each address before accessing it, and to hold all locks until it has finished writing. Furthermore, any other transaction T' that attempts to read an address written by T cannot do so until T has unlocked it. Therefore, T' cannot see any of the writes by T until all have been performed.

At a high level, the 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. The implementation ensures that the log is atomically flushed to NVM, so that it is accessible to the recovery process, *precisely* when it is committed. (Otherwise, committed transactions might be lost, or uncommitted transactions might be replayed by the recovery process.)

Detailed description. 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.

An *STMBegin* operation acquires *stmLock* as a reader (if the transaction has not exhausted its budget for attempts) or a writer (if it has). Acquiring *stmLock* as a writer prevents other transactions from running on the STM path.

An *STMRead* operation first invokes *GetLockAddr*(*addr*) to determine which *lock* in *locks* protects *addr*. Then, it invokes *TryReadLock*(*lock*) to acquire a read-lock, reads the address, and saves *addr* in its read-set. An *STMWrite* operation also starts by invoking *GetLockAddr*(*addr*) to identify the appropriate *lock*. It then 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 the write-log*. (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.) Next, *STMWrite* adds *addr* and *val* to its write-log entry. (Note

STM path

```
14 void STMBegin(log_entry* rec)
15   if budget of transaction attempts is exhausted then
16     WriteLock(stmLock)
17   else ReadLock(stmLock)
18 word STMRead(word* addr, log_entry* rec)
19   rwlock_t* lock = GetLockAddr(addr)
20   if !TryReadLock(lock) then
21     ResetLogEntry(rec)
22     unlock all locks
23     retry the transaction (goto line 15)
24   val = *addr
25   // remember addr so we can unlock it later
26   rset[rsize++] = addr
27   return val
28 // precondition: txn already invoked STMRead(addr, rec)
29 void STMWrite(word* addr, word val, log_entry* rec)
30   rwlock_t* lock = GetLockAddr(addr)
31   if !TryWriteLock(lock) then
32     ResetLogEntry(rec)
33     unlock all locks
34     retry the transaction (goto line 15)
35   // add <addr, val> to the write-log
36   rec->addr[rec->wsize] = addr
37   rec->wdata[rec->wsize] = val
38   rec->wsize++
39 bool STMFinalize(log_entry* rec)
40   FlushLogEntry(rec) // flush the log entry to NVM
41   // log entry is ready to be replayed
42   FetchAndIncrement(&numSTMWriteback)
43   rec->logged = 1
44   TFLUSH(rec->logged)
45   // replay the log entry to perform & flush all writes
46   ReplayLogEntry(rec, true /* perform writes */)
47   FetchAndDecrement(&numSTMWriteback)
48   unlock all locks
49   ResetLogEntry(rec)
50   rec->attempts = 0
51   return true
```

Slow STM path

```
52 void SlowHTMBegin(log_entry* rec)
53   if xbegin() == OK then   // transaction is started
54   else                     // abort handler
55     if budget of transaction attempts is exhausted then
56       move to STM path
57     else goto line 53      // try again
58 word SlowHTMRead(word* addr, log_entry* rec)
59   // check if addr is write-locked
60   rwlock_t* lock = GetLockAddr(addr)
61   if IsWriteLocked(lock) then xabort() // abort & retry
62   return *addr
63 void SlowHTMWrite(word* addr, word val, log_entry* rec)
64   // try to write-lock addr
65   rwlock_t* lock = GetLockAddr(addr)
66   if !TryWriteLock(lock) then xabort() // abort & retry
67   // add <addr, val> to the write-log
68   int wsize = rec->wsize
69   rec->wset[wsize] = addr
70   rec->wdata[wsize] = val
71   rec->wsize++
72   *addr = val // perform the write
73 void SlowHTMFinalize(log_entry* rec)
74   FlushLogEntry(rec) // flush the log entry
75   // atomically: commit in cache & set logged in NVM
76   xend(rec->logged) // (completing the log entry)
77   // replay the log entry to flush all writes
78   ReplayLogEntry(rec, false)
79   unlock all locks
80   ResetLogEntry(rec)
81   rec->attempts = 0
```

Fast HTM path

```
82 void FastHTMBegin(log_entry* rec)
83   if xbegin() == OK then   // transaction is started
84     if numSTMWriteback > 0 then xabort()
85   else                     // abort handler
86     if budget of transaction attempts is exhausted then
87       move to Slow HTM path
88     else goto line 83      // try again
```

Figure 2: Operations for each execution path

that *STMWrite* does not explicitly flush these changes, but they may trickle asynchronously to NVM.) If *STMRead* or *STMWrite* fails to acquire a lock, the transaction is aborted, all locks are released, and the transaction is retried.

To commit an STM transaction, a process invokes *STMFinalize*. *STMFinalize* first flushes the write-log entry to NVM, and then indicates that the transaction has entered its write-back phase by invoking *FetchAndIncrement* on *numSTMWriteback*. (We will see how *numSTMWriteback* is used in Section 4.4.) It then sets and flushes the *logged* bit in the log entry, which indicates that the log entry 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 4) 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.

4.3 The Slow HTM path

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 works in a hardware transaction because all writes remain in the processor's private cache until the transaction commits. Second, Slow HTM does not acquire any locks when it *reads* addresses. Instead, it simply *reads the state of the lock* for each of 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 it 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 *xend* 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.

Detailed description. A *SlowHTMBegin* operation invokes *xbegin* to start a hardware transaction, then verifies that it is executing in transactional mode (i.e., that *xbegin* returns *OK*). If not, then we jumped to this line in the code when our previous Slow HTM transaction aborted. So, *SlowHTMBegin* checks whether it has exhausted its budget of attempts. If so, the transaction moves to the STM path. Otherwise, *SlowHTMBegin* retries in hardware.

A *SlowHTMRead* operation reads the lock state for the address being read, and aborts (jumping to line 53) if another

process holds the lock as a writer. Otherwise, *SlowHTMRead* simply reads the address and returns the result. A *SlowHTMWrite* operation tries to lock an address 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 the write-log. If *SlowHTMWrite* acquires the lock, then it adds the address and the value to be written to the transaction's write-log entry. Finally, it performs the actual write.

To commit a transaction on Slow HTM, a process invokes *SlowHTMFinalize*. This flushes the write-log entry to NVM, and then invokes *xend*. Invoking *xend* 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 4) 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 were 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.

4.4 The Fast HTM path

Each transaction begins on Fast HTM. Writing and committing in Fast HTM transactions is the same as in Slow HTM transactions. Reads in Fast HTM transactions are more efficient, since they do not need to subscribe to locks to guarantee that the transaction sees a consistent state (i.e., that the addresses it read contained the values it saw at some point during the transaction). There are two reasons for this. First, the HTM system guarantees that transactions on Slow HTM cannot cause transactions on Fast HTM to see inconsistent state (or vice versa). Second, each transaction T on Fast HTM starts by verifying that *numSTMWriteback* is zero (and aborting, otherwise). As we saw in Section 4.2, *numSTMWriteback* is incremented whenever an STM transaction starts its write-back phase, and decremented whenever an STM transaction finishes its write-back phase. Thus, T will abort if it runs concurrently with any STM transaction in its write-back phase. Consequently, *FastHTMRead* is implemented as a simple read, with no additional synchronization.

FastHTMBegin first invokes *xbegin*, and then verifies that it is executing in transactional mode. Suppose it is. Then, *FastHTMBegin* verifies that *numSTMWriteback* is zero (aborting, otherwise), and returns. Suppose not. Then, we jumped to this line in the code when our previous Fast HTM transaction aborted. So, *FastHTMBegin* checks whether it has exhausted its budget of attempts. If so, the transaction moves to Slow HTM. Otherwise, *FastHTMBegin* tries again to execute the transaction on Fast HTM.

4.5 Executing on different paths

In this section, we describe when transactions on different paths can run concurrently. The results are summarized in Figure 3. Recall that each transaction on the STM path acquires *stmLock* as either a reader or writer. For convenience, we say that transactions which acquire *stmLock* as a reader (resp., writer) run on the *STM-R* (resp., *STM-W*) path.

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 3: Table showing which paths can run concurrently.

```
89  void FlushLogEntry(log_entry* rec)
90    TFLUSH(rec->wsize)
91    int wsize = rec->wsize
92    for i = 1..wsize
93      TFLUSH(rec->wset[i])
94      TFLUSH(rec->wdata[i])
95  void ResetLogEntry(log_entry* rec)
96    // prepare log entry for the next txn attempt
97    rec->lockfail = 0
98    rec->wsize = 0
99    rec->rsize = 0
100 void Recovery(int nprocesses)
101   unlock all locks for all processes
102   for i = 1..n
103     ReplayLogEntry(entries[i], true)
104 void ReplayLogEntry(log_entry* rec, bool doWrites)
105   if rec->logged then
106     int wsize = rec->wsize
107     if doWrites then
108       for i = 1..wsize // perform all writes
109         *rec->wset[i] = rec->wdata[i]
110     // transparently flush all writes
111     for i = 1..wsize
112       TFLUSH(*rec->wset[i])
113     // the log entry no longer needs replaying
114     rec->logged = 0
115     TFLUSH(rec->logged)
```

Figure 4: Functions common to all paths

A transaction T on Fast HTM can run concurrently with transactions on Slow HTM, since any conflicts are resolved by the HTM system. Additionally, T can run concurrently with any STM transaction, *as long as it is not executing its write-back phase* (denoted by Yes* in Figure 3). (If T is concurrent with an STM transaction in its write-back phase, then T might see only *some* of the writes performed by the STM transaction). Transactions on Slow HTM are always able to run concurrently with STM transactions, since both use locks. Since any transaction on STM-W acquires *stmLock* as a writer, it cannot run concurrently with any other transaction on STM-W or STM-R. (Note, however, that transactions on STM-W still acquire fine-grained locks, which is why they can *always* run concurrently with transactions on Slow HTM, and sometimes with transactions on Fast HTM.)

4.6 Recovery

After a power failure, the recovery process runs a simple procedure called *Recovery* (see Figure 4). 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* unlocks all processes' locks. (It can safely 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 processes 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 was committed, and its log was flushed to NVM. Next, the transaction's writes are performed at line 109. (Note that the recovery process performs these writes even for hardware transactions, despite the fact that

SlowHTMFinalize and *FastHTMFinalize* invoke *ReplayLogEntry* 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.) *ReplayLogEntry* then concludes by flushing all of the writes to NVM, and setting the *logged* bit to zero and flushing it to NVM.

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 the transaction's writes). Observe that the log entry's *logged* bit is reset to zero at the end of *ReplayLogEntry*. It follows that the power failure occurred before the process running this transaction could finish its invocation of ReplayLogEntry in *STMFinalize* (line 46), *SlowHTMFinalize* (line 78), or *FastHTMFinalize*. In each case, the process held write-locks on all addresses in the transaction's write-set.

4.7 Optimizing with non-transactional reads

In this section, we describe an optimization to PHyTM that can be applied in an HTM system that allows processes to perform non-transactional reads and writes while inside a transaction. This is currently possible only with the *tsuspend* and *tresume* instructions provided by IBM's implementation of HTM in its POWER7/8 processors, but other architectures are expected to provide support in the future.

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. [21] observed that it is sufficient to subscribe only to the lock, and to use a *non-transactional read* for the data protected by the lock. 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 would reduce the likelihood of capacity aborts.

5. CORRECTNESS

An execution history H of a transactional memory system is *opaque* [10] if one can choose a *serialization point* during each transaction T such that, if T's reads were all executed atomically at T's serialization point, they would return the same values as they do in H. (Note that, unlike serializability, opacity also requires *aborted* transactions to see a consistent view of memory up until they abort. This turns out to be a crucial safety property in transactional memory.) A transactional memory system is opaque if all possible histories are opaque. In this section, we prove that PHyTM is opaque. Due to space constraints, the progress proof is relegated to the full version of this paper [1].

We start with a few definitions. 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. A transaction attempt on Fast HTM **commits** at its execution of *xend*. A transaction attempt on Slow HTM **commits** at its execution of *xend* (at line 76). A transaction attempt on the STM path **commits** at its execution of *TFLUSH* (at line 44).

We now give serialization points for all transactions. In this section, we suppose no power failures occur, and prove that PHyTM is opaque with these serialization points. In the next section, we consider power failures.

Serialization points

- Each committed transaction is serialized precisely when it *commits* (see the definition above).
- Each aborted transaction attempt on Fast HTM is serialized at the last time it accesses a lock in *FastHTMWrite*.
- Each aborted transaction attempt on Slow HTM is serialized at the last time it accesses a lock in *SlowHTMRead* (line 61) or *SlowHTMWrite* (line 66).
- Each aborted transaction attempt on the STM path is serialized at the last time it accesses a lock in *STMRead* (line 20) or *STMWrite* (line 31).

LEMMA 1. *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 36-38), SlowHTMWrite (lines 69-71) or FastHTMWrite.*
3. *T locks addr as a writer (in STMWrite at line 31, in SlowHTMWrite at line 66, 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 109 of *ReplayLogEntry*. Prior to invoking *ReplayLogEntry* at line 46, it commits at line 44 (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 72 (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 66. 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 79, which is after T flushes addr to NVM in its invocation of *ReplayLogEntry* (at line 78 of *SlowHTMFinalize*), which is after T commits in *SlowHTMFinalize* at line 76. The case where T executes on Fast HTM is proved similarly. □

LEMMA 2. *Let T be a transaction attempt with addr in its write-set that commits at time t_c, and t_u be when T releases its write-lock on addr. Starting from some time t_v ($t_c \leq t_v \leq t_u$), addr continuously contains the last value v written by T until a write-lock is next acquired on addr after t_u by a transaction attempt that commits.*

PROOF. Let t_w be when T writes to *addr* and t_f be when T flushes *addr* to NVM. By Lemma 1, 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 1). □

LEMMA 3. *Let R be an invocation of STMRead($addr, rec$) by a transaction attempt T. If R returns at line 27 (as opposed to causing the transaction retry at line 23), then R returns the value v written by the last committed transaction T', with addr in its read-set, that is serialized before T.*

PROOF. Let t_r be when R executes line 24. 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 2, *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 20 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 2, *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 1, T'' must commit before T acquires its write-lock on *addr*, which implies that T'' is committed between T' and T (and, hence, is serialized between T' and T). However, we assumed that T' is the last committed transaction with *addr* in its write-set that is serialized before T. □

LEMMA 4. *Let R be an invocation of SlowHTMRead(addr, rec) by a transaction attempt T. If R returns at line 62 (as opposed to aborting the transaction at line 61), then R returns the value v written by the last transaction T', with addr in its write-set, that is serialized before T.*

PROOF. Although R does not explicitly acquire read-locks, it reads 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 subscribe to it, T will abort if any concurrent transaction attempt acquires the lock after it is read by R.) Thus, the value it reads at line 62 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 3. □

LEMMA 5. *Let R be an invocation of FastHTMRead(addr, rec) by a transaction attempt T. R returns the value v written by the last transaction T', with addr in its write-set, that is serialized before T.*

PROOF. Let t_r be when R reads *addr*. 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 2, *addr* contains v at some time after T' commits, and before it releases its locks. Let t_v be the earliest such time. 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, which is a contradiction.

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 2, *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 1, T'' must commit before T acquires its write-lock on *addr*, which implies that T'' is committed between T' and T (and, hence, is serialized between T' and T). However, this is a contradiction, since we assumed that T' is the last committed transaction, with *addr* in its write-set, that is serialized before T. □

Together, Lemmas 3, 4 and 5 imply that each read performed by a transaction T returns the value written by the last committed transaction serialized before T. Consequently, T performs the same sequence of steps as it would in the serialized execution. Therefore, PHyTM is opaque.

5.1 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.

We start with a definition. A transaction attempt is **logged** when the *logged* bit in its log entry is set in NVM. Observe that a committed transaction attempt is serialized, 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 6. *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 7. *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 serialized) before a power failure. Suppose T does not terminate 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 also logged before the power failure. □

THEOREM 8. *Immediately after the recovery process finishes executing the Recovery procedure, the contents of NVM 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 and flush them to NVM.

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 6 implies that all logged transactions operate on disjoint write-sets. □

6. RELATED WORK

Non-volatile RAM is expected to replace DRAM, either partially or entirely, as main memory [17]. There are already working prototypes of NVM such as phase-change memory (PCM) [17], spin-torque-transfer RAM (STT-RAM) [12], and memristors [23], and the new Intel architecture added special instructions (CLFLUSHOPT, CLWB) [13] 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 [5] and Mnemosyne [24] 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 [24], which was published at the same time as NV-Heaps, is word-based, and is derived from TinySTM [9]. 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 [25, 18] use HTM for synchronization. However, these databases still use flush data to disk to achieve persistence.

In [26], Wang et al. propose 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. Whereas [26] 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 fast path concurrency with the slow path were introduced in 2006 [7, 16]. Since then, research on hybrid TM algorithms has been focused on optimizations to improve performance.

Optimizations for hybrid TMs have moved 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 [6]. (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 [19].)
- Attaching a (versioned or traditional) lock to each address, which is read by each HTM read operation, and is acquired on both paths for writes, in the HyLSA algorithm [21]. This 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 low overhead, but is unscalable, so we used a similar approach to HyLSA in PHyTM. The use of three execution paths to improve concurrency in PHyTM is based on the work of Brown [3].

As we discussed in Section 4.7, numerous accesses in PHyTM can be non-transactional. Since both persistent HTM and STM transactions acquire locks on each address in their write-sets, 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 guards). Additionally, since each read on the STM path 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.

7. EXPERIMENTAL ANALYSIS

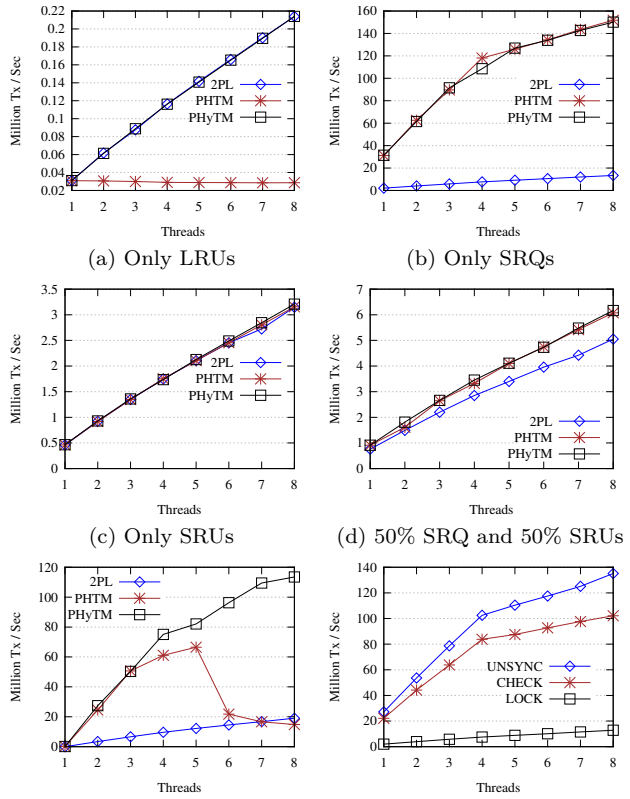
In this section we study how PHyTM can be used to reduce synchronization costs for simple IMDBs.

Workloads. We implemented a very simple IMDB (VSDB), and used it to run a subset of the Yahoo! Cloud Serving Benchmark (YCSB). We also studied the TPC-C benchmark using DBx1000, the IMDB implementation from [27]. Our simple YCSB benchmarks demonstrate the low synchronization cost of PHyTM, and the TPC-C benchmark illustrates its performance with more complex transactions.

Synchronization methods. We modified each IMDB to use several different synchronization methods. In both IMDBs, we added support for PHyTM and PHTM. In VSDB, we implemented a simple 2PL scheme which performs fine-grained locking on rows in encounter order. Our YCSB workloads cannot cause deadlock, so we did not implement deadlock detection for 2PL. DBx1000 already featured 2PL and optimistic concurrency control (OCC) as synchronization methods. Their 2PL implementation performs fine-grained locking on rows, and incorporates deadlock detection. These 2PL and OCC implementations were shown to be scalable in simulations with more than one thousand processors [27].

System. We use an Intel i7-4770 3.4 GHz processor with 4 cores, each with 2 hyperthreads. Each core has a private 32KB L1 cache and a private 256KB L2 cache, and an 8 MB L3 cache is shared by all cores. We use the HTM provided by the hardware and emulate NVM. In our experiments, threads are pinned so one thread runs on each logical processor.

Emulation. We emulate NVM support using the approach taken in [2]. The atomic assignment and flush of the



(a) Only LRUs (b) Only SRQs

(c) Only SRUs (d) 50% SRQ and 50% SRUs

(e) One thread does LRQs, and the rest do SRQs (f) How much HTM reduces locking costs in PHyTM

Figure 5: YCSB workloads for different transaction mixes.

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 for all synchronization methods.

7.1 YCSB

In all of our YCSB experiments, we used a single table with 20 million records and a single primary key column. We perform four types of transactions: *short range queries* (SRQs), *long range queries* (LRQs), *short range updates* (SRUs) and *long range updates* (LRUs). The *short* (resp., *long*) transactions lookup 16 (resp., 256) random keys in the table. The *query* (resp., *update*) transactions select a column (resp., write to a column). These transactions simply read or write the contents of columns, and do not perform any additional computation. As an example, SQL code for a *query* transaction might be "SELECT name FROM customers WHERE id IN (1350, 2107, ..., 571)." SQL code for an *update* transaction might be "UPDATE customers SET orders=7 WHERE id=1350 ; UPDATE customers SET orders=4 WHERE id=2107 ; ..." All transactions are independent. That is, the behaviour of a transaction does not depend on the result of a previous transaction. Note that, since *long* transactions access a fairly large number of rows, they are somewhat likely to cause capacity aborts.

Results. The results appear in Figure 5. 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 algorithms 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 similar performance in write-only scenarios, as Figure 5c shows. However, Figure 5b shows that reads in PHTM and PHyTM an order of magnitude faster than in 2PL.

Figure 5d shows a YCSB workload that performs 50% SRQs and 50% SRUs. 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 because of capacity aborts. PHTM includes an STM fallback path to allow large 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 LRUs, we see in Figure 5a 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 many expensive writes/flushes to NVM until after it commits. Thus, aborted transactions avoid this overhead.

In the workload consisting entirely of LRUs, PHyTM and 2PL achieve approximately the same throughput, and perform an order of magnitude better than PHTM. In the workload consisting entirely of SRQs, 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 small number of threads are running transactions on the STM path while most threads are successfully committing transactions in HTM. This situation is demonstrated in Figure 5e, where one thread is executing LRQs (which are unlikely to succeed in hardware, and often run on the STM path), and the other threads execute SRQs that are typically able to commit in hardware. In this case, PHyTM is an order of magnitude faster than its competitors with eight threads. 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 threads, but suffers from the high cost of acquiring locks. Whenever there is no STM transaction *in its write-back phase*, PHyTM transactions can run on Fast HTM, where their read operations have no overhead. Furthermore, even when a transaction on the STM path is in its write-

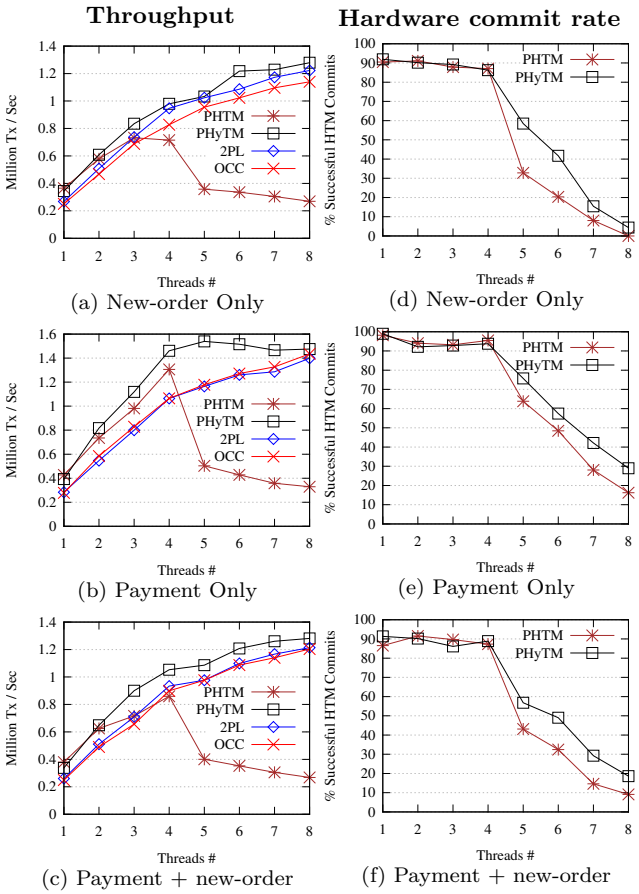


Figure 6: TPC-C workloads with 8 warehouses.

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

In order to show how much HTM reduces locking synchronization costs in PHyTM, we ran the read-only workload shown in Figure 5b with three different variants of PHyTM. In the first variant, *LOCK*, reads on all paths acquire locks (just like the STM path). In the second variant, *CHECK*, reads on all paths simply check the state of the lock (just like the Slow HTM path). In the third variant, *UNSYNC*, reads on all paths are simply uninstrumented reads (just like the Fast HTM path). (Note that these algorithms are correct only for a read-only workload.) The results, which appear in Figure 5f, show that checking lock state is more than 5x faster than acquiring a lock, and uninstrumented reads are approximately 20% faster than checking the lock state.

7.2 TPC-C

We used the TPC-C implementation provided by DBx1000. It implements the *new-order* and *payment* transactions, which comprise 88% of transactions executed in the full TPC-C benchmark. At a high level, new-order transactions insert several rows into a table, and also access rows of three other tables. Payment transactions update several rows of a table, insert rows into another table (*history*), and access rows of three other tables.

PHyTM, like all HTM-based algorithms, significantly improves performance only if most hardware transactions commit. New-order and payment transactions can have relatively large data-sets, and often cause capacity aborts. One way to reduce the likelihood of a transaction experiencing a capacity abort is to use a transaction chopping algorithm to decompose the transactions into smaller pieces [4]. (This requires analyzing transactions dependencies to determine where transactions can be split.) For simplicity, we chose to shrink the transactions so they are better suited to current hardware limitations: For payment transactions, we skip the insertion of rows into the *history* table, and for new-order transactions, we reduce the number of rows inserted. Future generations of Intel's HTM are expected to have larger transaction capacities [14].

Results. Our workloads perform different mixtures of payment and new-order transactions with eight warehouses. For each workload, we created a graph showing transaction throughput, and a graph showing the percentage of transactions that completed successfully in hardware (Figure 6). As expected, as the fraction of transactions succeeding in hardware decreases, PHyTM performs increasingly better than PHTM. This is in spite of the fact that the difference in the fraction of transactions succeeding in hardware between PHyTM and PHTM is relatively small. For example, at eight threads in Figure 6a, PHyTM performs 4x as many transactions as PHTM, but transactions in PHyTM succeed in hardware only 11% more often than in PHTM.

Recall that, each time a transaction aborts, the HTM system reports an *abort reason* that describes why the abort occurred. We inspected the reasons for aborts in PHyTM and PHTM, and saw that PHyTM and PHTM experience a similar number of capacity aborts, but PHTM experiences many more conflict aborts. This is because PHTM subscribes to a global lock at the beginning of each hardware transaction, and, whenever the global lock is acquired, all concurrent hardware transactions experience conflict aborts.

The 4x difference in performance between PHyTM and PHTM in Figure 6a is a result of threads being serialized by the global lock acquired on PHTM's fallback path. When the thread count exceeds 4, the number of capacity aborts increases due to hyperthreading. In PHTM, this causes the fallback path to be executed more often, which dramatically decreases concurrency. However, in PHyTM, hardware transactions can run concurrently with software transactions, so PHyTM manages to commit more transactions in hardware.

All TPC-C transactions include writes and additional computation, so the synchronization overhead is less significant than in the YCSB. The advantage of PHYTM over OCC and 2PL is a function of synchronization overhead and HTM success rate. We can see PHyTM performs better for payment transactions (Figure 6b) than in the workloads with new-order transactions. This is because new-order transactions perform more computation (so synchronization is a smaller factor), and commit less often in hardware.

8. CONCLUSION

Efficient, persistent hybrid TM will allow 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.

Acknowledgments

Hillel's work was fully funded by Huawei's Central Software Institute (CSI). Trevor's work was funded by the National Science and Engineering Research Council of Canada.

9. REFERENCES

[1] H. Avni and T. Brown. Persistent hybrid transactional memory for databases. Manuscript available from `http://phytm.tbrown.pro`.

[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] T. Brown. Brief announcement: Faster data structures in transactional memory using three paths. In *Proceedings of the 25th ACM Symposium on Distributed Computing*, pages 671–672. Springer-Verlag Berlin Heidelberg, 2015.

[4] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the 12th European Conf. on Comp. Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 26:1–26:17, 2016.

[5] 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 16th International Conference*, ASPLOS, pages 105–118, New York, NY, USA, 2011. ACM.

[6] 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.

[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. of 12th Int. Conf. on Arch. Supp. for Prog. Lang. and Oper. Sys., ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 336–346, 2006.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Distr. Comp.*, volume 4167 of *Lec. Notes in Comp. Sci.*, pages 194–208. Springer Berlin Heidelberg, 2006.

[9] 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.

[10] 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.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of 20th Int. Sym. on Comp. Arch.*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

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

[13] Intel. Intel architecture instruction set extensions programming reference.

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

[15] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with intel® transactional synchronization extensions. In *2014 IEEE 20th Int. Sym. on High Perf. Comp. Arch. (HPCA)*, pages 476–487. IEEE, 2014.

[16] 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.

[17] 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.

[18] 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.

[19] A. Matveev and N. Shavit. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *Proc. of the 20th Int. Conf. on Arch. Supp. for Prog. Lang. and Oper. Sys.*, ASPLOS '15, pages 59–71, New York, NY, USA, 2015. ACM.

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

[21] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: the importance of nonspeculative operations. In *SPAA 2011: Proc. of the 23rd ACM Sym. on Par. in Alg. and Arch., San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 53–64, 2011.

[22] D. Schwalb, M. Faust, M. Dreseler, P. Flemming, and H. Plattner. Leveraging non-volatile memory for instant restarts of in-memory database systems. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1386–1389. IEEE, 2016.

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

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

[25] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys '14, pages 26:1–26:15, New York, NY, USA, 2014. ACM.

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

[27] 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.