PhTM: Phased Transactional Memory *

Yossi Lev
Brown University and
Sun Microsystems Laboratories
yosef.lev@sun.com

Mark Moir Dan Nussbaum
Sun Microsystems Laboratories
{mark.moir,dan.nussbaum}@sun.com

Abstract

Hybrid transactional memory (HyTM) [3] works in today's systems, and can use future "best effort" hardware transactional memory (HTM) support to improve performance. Best effort HTM can be substantially simpler than alternative "unbounded" HTM designs being proposed in the literature, so HyTM both supports and encourages an incremental approach to adopting HTM.

We introduce Phased Transactional Memory (PhTM), which supports switching between different "phases", each implemented by a different form of transactional memory support. This allows us to adapt between a variety of different transactional memory implementations according to the current environment and workload. We describe a simple PhTM prototype, and present experimental results showing that PhTM can match the performance and scalability of unbounded HTM implementations better than our previous HyTM prototype when best effort HTM support is available and effective, and is more competitive with state-of-the-art software transactional memory implementations when it is not.

1. Introduction

The multicore revolution currently in progress is making it increasingly important for applications to be concurrent in order to take advantage of advances in technology. But today's concurrent programming practice, mostly based on locks and condition variables, is inadequate for this task. Locks do not compose, and introduce troublesome tradeoffs between complexity, performance, and scalability. Furthermore, locks are subject to deadlock if not used carefully, resulting in significant software engineering problems.

Transactional Memory [9] is widely considered to be the most promising avenue for addressing these problems. Using transactional memory, programmers specify *what* should be done atomically, rather than specifying *how* this atomicity should be achieved, as they do today with locks. The transactional memory implementation guarantees atomicity, largely relieving programmers of the above-mentioned tradeoffs and software engineering problems.

Transactional memory can be implemented in hardware [9], in software [16], or in a combination of the two [3, 14, 15]. Hardware transactional memory (HTM) designs can be unbounded, bounded, or "best effort". The original HTM proposal [9] is a bounded HTM: it has a fixed-size, fully-associative transactional cache, and a transaction can be committed if and only if it fits in that cache.

Alternative "best effort" designs, such as those that piggyback on existing caches and other hardware structures such as store buffers [18], may be able to commit one large transaction while being unable to commit another significantly smaller one, depending on how the transactions happen to map to the existing structures. Best effort HTMs are not required to make particular guarantees about what transactions can commit, and are therefore substantially easier to design, for example because difficult corner cases can be handled by simply aborting a transaction.

Recently, numerous proposals for "unbounded" HTM [2, 8, 14, 15] have appeared in the literature in an effort to overcome the shortcomings of bounded and best effort HTM designs. However, all of them entail substantially more complexity than the much simpler best effort implementations, and therefore it will be much more difficult to integrate unbounded HTM solutions into commercial processors in the near future. We believe that simple best effort HTM support can be implemented much sooner. However, if used directly, best effort HTM support can impose unreasonable constraints on programmers, for example requiring them to think about the number and distribution of cache lines accessed by transactions or other architecture-specific implementation details. These tradeoffs are a large part of the reason HTM has not been embraced by the computer industry to date.

Software transactional memory (STM) [16] provides software engineering benefits similar to those of HTM, but does not require special hardware support, and thus can be built and used in existing systems today. Because it is independent of hardware structures such as caches and store buffers, STM is not subject to the limitations of bounded and best effort HTMs. Substantial progress in improving STM in various ways has been made in recent years. Nonetheless, it

remains much slower than what can be expected from HTM, and a substantial gap will likely always remain.

We previously proposed Hybrid Transactional Memory (HyTM) [3], which aims to provide the flexibility and generality of STM, but also to exploit HTM support *if it is available and when it is effective* to boost performance. A HyTM implementation minimally comprises a fully-functional STM implementation, because it must work even if there is no HTM support available. Thus, any transaction can be executed in software, without special hardware support. This allows us to develop, test, and run transactional programs in existing systems, even before any HTM support is available. Furthermore, a HyTM implementation can use best effort HTM support to execute transactions if it is available. This way, significant performance improvements are possible with HTM support if many transactions can be committed by the HTM, even if some must be executed in software due to limitations of the HTM support.

In the prototype HyTM system described in [3], transactions executed using HTM augment every transactional load and store with code to check for conflicts with concurrent software transactions. Simulation studies we have conducted show that while HyTM can perform dramatically better with HTM support than without, a simulated unbounded HTM implementation performs significantly better still, because it does not have to pay the overhead for detecting conflicts with concurrent software transactions. Furthermore, the need for transactions executed in software to expose enough information to make it possible for transactions executed using HTM to detect conflicts imposes overhead on and constrains the design of the STM component of HyTM, making it difficult to compete with other STMs that do not have this requirement. For example, our HyTM prototype used in software-only mode is outperformed by state-of-the-art STMs, e.g., TL2 [4].

In this paper, we introduce Phased Transactional Memory (PhTM). Our goal is to build a HyTM system in which transactions successfully executed using best effort HTM perform (almost) as well as an unbounded HTM would, while transactions executed in software are competitive with the fastest STM systems.

Phased Transactional Memory (PhTM)

The key idea behind PhTM is to support various modes that are optimized for different workload and system support scenarios, and to allow seamless transitions between these modes. The name Phased Transactional Memory comes from the idea that we might operate in one mode for some time, then switch to another, thus executing the application in "phases" in which the system is in a particular mode for each phase. The challenges include:

- identifying efficient modes for various scenarios
- managing correct transitions between modes
- deciding when to switch modes

In this paper, we present options for addressing each of these issues, describe a prototype we have implemented to evaluate the potential of this approach, and present performance experiments showing that PhTM is a promising approach. While the PhTM approach is applicable to other contexts, our prototype is integrated into a C/C++ compiler based on the one described in [3].

The rest of this paper is organized as follows: Section 2 presents a range of implementation alternatives for PhTM. Section 3 describes our prototype implementation. Section 4 presents experimental results and analysis. We conclude in Section 5.

2. A General PhTM Framework

The key idea behind PhTM is to use multiple, mostly independent transactional memory implementations, and to support switching between them seamlessly. This way, we can use the mode best optimized for the current workload and execution environment, but switch to another in response to changes in the workload or if a transaction that is not supported by the current mode is encountered. There is a wide variety of possible implementations that could be used in a PhTM system. In this section, we describe some common scenarios and TM implementations that are appropriate for them, and discuss how we can integrate and switch between them. We structure our presentation by focusing on the challenges described above.

2.1 Identifying efficient modes for various scenarios

Below we identify several likely scenarios and describe execution modes that are effective for them.

1. HTM is available and almost always effective.

   For this scenario, we propose a `HARDWARE` mode in which all transactions are executed using HTM support. Because there is no need to detect conflicts with concurrent software transactions, we can eliminate the substantial overhead in our existing HyTM prototype for checking for such conflicts.

2. HTM is unavailable or is ineffective for the current workload.

   For this scenario, there is little point in attempting to execute transactions using HTM support. Therefore, in this case we use a `SOFTWARE` mode, in which all transactions are executed in software. This way, the STM design is not constrained to interoperate correctly with HTM transactions, which means we can use state-of-the-art optimized STMs that have not been designed to interoperate with HTM.[1]

[1] Not all STMs can be used in this way. In particular, STMs that require transactional metadata to be co-located with transactional data (e.g., [7]) cannot be integrated into a C/C++ compiler, because they interfere with standard data layout.

3. HTM is available, and is usually effective but often not.

   In this scenario, it may be worthwhile to execute HTM-supported and software transactions concurrently in order to avoid frequent mode changes. In this case the HTM transactions and software transactions must ensure that conflicts between them are correctly detected and resolved. In this `HYBRID` mode, the system behaves like our current HyTM prototype [3].

4. Workload is single-threaded, or uses few transactions.

   This mode (call it `SEQUENTIAL`) supports the execution of only one transaction at a time. The thread that causes a transition into this mode can execute its transaction with no conflict detection because there are no concurrent transactions, which can eliminate a significant amount of overhead.

5. Workload is single-threaded, or uses few transactions *and* some transactions are known not to abort explicitly.

   For such transactions, we can use a `SEQUENTIAL-NOABORT` mode that essentially eliminates virtually all transactional memory overhead. As in the previous case, the thread that causes a transition into this mode can execute its transaction with no conflict detection. However, because transactions will not abort due to conflict and will not abort explicitly, this mode can also elide logging, making the resulting execution essentially the same as sequential execution. This mode also allows general functionality that might not be supported in other modes, for example executing I/O or system calls within transactions.

We believe that many practical scenarios will fall into category 1 or 2 above. If we were always to use our HyTM prototype as it exists today (which essentially always operates in `HYBRID` mode), we would be unnecessarily penalizing all such applications, in effect making them pay overhead to support unneeded flexibility. By supporting multiple modes we are able to use the best mode for each given environment and workload. Furthermore, by supporting dynamic mode changes, we can adapt to changing workload characteristics, or make optimistic assumptions and transparently back out of them if they turn out to be wrong.

2.2 Managing correct transitions between modes

A key challenge in implementing PhTM is in ensuring that we can change modes effectively when we decide to do so. Because different modes support different transaction implementations, it is necessary when switching modes to prevent transactions executing in the current mode from compromising the correctness of the transactions executing in the new mode, and vice versa. One simple approach is to ensure that all transactions executing in the current mode are either completed or aborted before allowing transactions in the new mode to begin. This can be achieved in various ways, depending on the old and new modes.

In `HARDWARE` mode (scenario 1 above), it is sufficient to check once per transaction that the system is in a mode that allows HTM transactions that do not detect conflicts with software transactions. This can be done, for example, by reading (as part of the transaction) a global `modeIndicator` variable, and confirming that the system is in a mode compatible with HTM-only execution. Having performed this check, it is safe to execute the rest of the transaction without any additional overhead for conflict checking because either a) the transaction will complete while the mode remains unchanged, or b) `modeIndicator` will change, thus causing the transaction to fail. This checking entails very low overhead because it is a single load of a variable that is read-only until the next mode change, and this check is performed once per transaction, not once per memory access as are the conflict checks in our current HyTM prototype. Transactions executed using HTM in `HYBRID` mode (scenario 3 above) can use the same technique to ensure they execute in a compatible mode, but additionally need to check for conflicts with software transactions.

The situation is more complicated for software transactions. Before switching from a mode that allows software transactions to a different mode, we must ensure that each transaction currently executing in software can no longer do anything to compromise the correctness of transactions to be executed in the next mode.

One simple approach is to prevent new software transactions from starting, and to wait for all software transactions already executing to complete. Alternatively, in some cases it may be preferable to abort some or all transactions executing in the current mode in order to facilitate faster mode switches. This might be achieved by iterating over all active software transactions and explicitly aborting them, or alternatively by a "safe point" mechanism [1] in which, with compiler support, every transaction is always guaranteed to check within a short time whether there has been a mode change and therefore it should abort itself.

2.3 Deciding when to switch modes

Given the ability to switch between modes, a wide range of policies for deciding when to switch modes and to which mode is possible. For a simplisitic example, we could "schedule" phases, spending a fixed amount of time in each mode in order. But we can likely improve performance in most cases by monitoring progress of transactions, commit/abort rates, status of transactions with respect to the current mode, etc. For example, if we find that almost all transactions succeed in `HARDWARE` mode, we will generally prefer to spend most time in that mode. But if we encounter a transaction that we know or suspect can *never* commit in `HARDWARE` mode, for example due to limitations of the HTM support, then we must eventually switch to a mode that can execute the transaction in software. Once such a transaction is encountered, we may choose in some cases to wait for a short time to allow other transactions that require a

SOFTWARE mode to accumulate, or we might switch immediately to a SOFTWARE mode to avoid delaying the transaction.

Similarly, when we are in a mode that supports software transactions, we may wish to switch back to HARDWARE mode in order to again get the performance benefit of executing transactions using HTM without checking for conflicts with software transactions. Again, a number of factors, including programmer hints about the application or statistics collected perhaps by a contention manager may influence this decision. We would probably wish to minimally wait for completion of the transaction that initiated the switch to a SOFTWARE mode before switching back to HARDWARE mode, in order to avoid "thrashing" between modes (because such a transaction, if reexecuted in HARDWARE mode, is likely to immediately initiate a return to a SOFTWARE mode).

2.4 Example PhTM implementation

For concreteness, in the remainder of this section we describe an example PhTM implementation that can support a number of different modes. Before a thread begins a transaction, it checks the current mode. It can then either attempt to execute in that mode or choose to change to a different mode, depending on policy decisions driven by a host of potential factors such as knowledge of the application or transaction (provided by the programmer, compiler or command line options, or profiling data), knowledge of the success or otherwise of previous attempts in this mode, etc. We first explain how mode changes are supported, and then explain how transactions are executed in various modes.

Mode-changing mechanism

In general it is not safe to change modes without waiting for some condition to hold, and in some cases it is desirable to delay mode changes in order to avoid "thrashing" between modes, as well to ensure that threads that wanted to switch to a new mode can take advantage of the new mode before the mode changes again. Below we present a general mechanism that supports these purposes.

For simplicity of exposition, we base our mode changing mechanism on a single, globally shared modeIndicator variable, which contains 6 fields, as follows:

modeIndicator = ⟨ mode, mustFinishThis, otherTxns, nextMode, mustFinishNext, version⟩

We assume the ability to atomically update modeIndicator, for example by storing its fields in a 64-bit word and using a 64-bit compare-and-swap (CAS) instruction to modify it. We later explain some potential shortcomings of this simple approach, and how to overcome them. The fields in modeIndicator are described below.

mode indicates the current mode, e.g., HARDWARE, SOFTWARE, HYBRID, SEQUENTIAL, SEQUENTIAL-NOABORT.

mustFinishThis is the number of transactions that must complete in this mode before we switch to the next mode.

otherTxns is used in some modes to count transactions that are in this mode but are *not* included in the mustFinishThis counter; this field may be used in modes that require all transactions executing in that mode to indicate that they have completed or aborted before the next mode switch.

nextMode indicates the next mode to switch to; it contains NONE when no mode transition is in progress.

mustFinishNext is the number of transactions involved in switching to the next mode; these will be the transactions that must complete in nextMode mode (as opposed to others that may be aborted) after the next mode switch.

version is a version number that is incremented with every mode change, to avoid the so-called ABA problem, in which a transaction twice observes the same value in all other fields of modeIndicator, and incorrectly concludes that modeIndicator did not change in between the two observations.[2]

Initially, mode is the default starting mode (e.g., HARDWARE), nextMode is NONE, and all other fields are zero.

If no mode change is in progress (i.e., nextMode is NONE), then a thread can *initiate* a mode change by changing nextMode to the desired mode, while keeping all other fields unchanged (with the possible exception of setting mustFinishNext to one; see below). Subsequently, perhaps after a short delay, this thread or some other thread can *complete* the mode change, provided mustFinishThis and otherTxns are both zero, as follows:

- copy nextMode to mode
- set nextMode to NONE
- copy mustFinishNext to mustFinishThis
- set mustFinishNext to zero
- increment version

Between the initiation and completion of a mode change, threads that desire the same mode change can *join* the mode change, by incrementing mustFinishNext. This ensures that such threads are included in the mustFinishThis counter after the mode change, so the mode does not change again until these threads complete their transactions and decrement mustFinishThis.

In some modes, some transactions cannot be completed, for example because they use functionality not supported by the current mode. In such cases, in order to allow subsequent mode changes, a thread joining a mode change to a mode that is not guaranteed to be able to complete its transaction should either refrain from incrementing mustFinishNext,

[2] The version number may not be strictly necessary in the example described here, but nonetheless makes reasoning about the algorithm easier. Furthermore, implementations that separate the mode indication from other components of the algorithm (as described later) will likely require a version number, e.g., to prevent "old" mode change attempts from succeeding.

or guarantee that, after the mode change, it will eventually decrement mustFinishThis, even if it does not complete.

Execution modes

Next we describe how transactions execute in each mode.

HARDWARE mode: in this mode, transactions begin a hardware transaction, read modeIndicator and confirm that it still indicates HARDWARE mode (aborting if not), and then execute the transaction in the hardware transaction, without explicitly checking for conflicts with other transactions. This is safe because in this mode all transactions are executed using HTM support, so conflict detection and resolution is handled in hardware. If the mode changes before the transaction completes, the transaction will abort because it has read modeIndicator.

SOFTWARE mode: in this mode, if the thread did *not* increment mustFinishNext during the transition to this mode, then it must increment the otherTxns counter while confirming that the mode is still SOFTWARE. In some implementations, if nextMode is not NONE or mustFinishThis is zero, the thread waits until the mode changes (or changes the mode itself). This technique may be used to encourage a change back to a more efficient mode (e.g. HARDWARE) after the transactions for which we changed to SOFTWARE mode have completed.

With some STM algorithms we might choose to only try to increment the otherTxns counter before the transaction enters the time period in which it is unsafe to switch modes (for example, immediately before committing in the STM of our HyTM prototype). In this case, if we fail to increment the counter, the transaction must abort and be retried in the next mode. By delaying the increment of otherTxns until we are about to do something incompatible with the next mode, we can reduce the time we must wait to switch modes. Furthermore, when using a write set approach as the STM of our HyTM prototype does, this ensures that we do not wait for these "other" transactions while they are executing user code, which may take much longer and be less predictable than the commit code, which is under our control.

Once a thread has incremented otherTxns, the mode will not change before it decrements otherTxns, which it will do only when it is guaranteed not to perform any action that jeopardizes correctness if the mode changes (for example after it has committed successfully, or has aborted). This makes it safe for the thread to execute its transaction using a state-of-the-art STM, regardless of the details of the inner workings of that STM.

Because no hardware transactions execute in SOFTWARE mode, there is no need to facilitate the detection of conflicts between hardware and software transactions, so the design of the STM is essentially unaffected by its inclusion in our PhTM system. There are, however, some small changes necessary, which are applicable to any STM system. Specifically, upon completion, a transaction must report its completion, either by decrementing the mustFinishThis counter (in case it incremented mustFinishNext during the transition to the current mode) or the otherTxns counter (otherwise). Furthermore, to facilitate faster mode changes, it may be desirable for transactions to abort before completion upon observing that a mode change has been initiated (i.e., nextMode is not NONE). This can be achieved in a number of possible ways, for example checking modeIndicator before retrying, upon executing a transactional load or store, on backwards branches, or at compiler-supported safe points [1], etc. Once the transaction has aborted, it decrements otherTxns to reflect this.

HYBRID mode: In this mode, transactions can be attempted using either hardware or software transactions, each modified to facilitate detection and resolution of conflicts with the other. For example, in our HyTM prototype [3], hardware transactions are augmented to detect conflicts with software transactions, and software transactions maintain state sufficient to facilitate this.

To allow integration with the PhTM system, HYBRID transactions executed in hardware must additonally read modeIndicator once at the beginning to confirm the HYBRID mode (as described above for HARDWARE mode) in order to ensure that they abort upon mode change. Similarly, transactions executed in software coordinate using the mustFinishThis and otherTxns counters as described above for SOFTWARE mode. This way, we can ensure that no transaction (hardware or software) executed in HYBRID mode interferes with transactions executed in subsequent modes.

SEQUENTIAL and SEQUENTIAL-NOABORT modes: These modes are similar to SOFTWARE mode, except that they can use "stripped down" STMs to improve performance: neither mode requires conflict detection, and SEQUENTIAL-NOABORT does not require logging for abort either, and can thus execute almost at the speed of sequential code.

Enhancements to the simple scheme

Because hardware transactions read modeIndicator to confirm that the system is executing in a compatible mode, changes to other fields of modeIndicator may cause them to abort. Therefore, it may be preferable to *coordinate* mode changes using different variable(s), and just use modeIndicator to actually change modes. By keeping version numbers in these other variables coordinated with the version field in modeIndicator, it is not difficult to arrange for all transitions other than actual mode changes to affect other variables, so that only mode changes modify the modeIndicator variable that indicates the current mode.

The counters for mustFinishThis, mustFinishNext, and otherTxns may introduce bottlenecks that inhibit scalability past a certain number of threads or cores if implemented naively. Fortunately, we observe that these counters do not generally need to provide the ability to know the value of the counter, as provided by standard counters. It is sufficient that they support Arrive, Depart, and

Query operations, where the Query operation simply reports whether there is any transaction that has arrived and not yet departed. "Scalable Non-Zero Indicators" (SNZI) [6] are designed exactly for this purpose. SNZI exploits the fact that the required semantics is weaker than for standard counters to allow scalable and efficient implementations. The SNZI mechanism specifically targets implementations in which the Query operation comprises a single read of a variable that will not change before the number of transactions that have arrived and not yet departed becomes zero or becomes nonzero. Thus, by using SNZI objects in place of the above-mentioned counters, we can effectively integrate the above-described mode coordination scheme with hardware schemes without unrelated events causing aborts of hardware transactions.

3. Our experimental prototype

There is a virtually endless supply of modes that may make sense for some scenario, as well as a wide variety of potential ways to decide when to switch modes and to which mode. But in practice we prefer to have a small set of modes and simple but effective policies for switching between them. We are convinced that we would at least like to have a HARDWARE mode that is as fast and scalable as possible when all transactions can be committed using HTM, and a SOFTWARE mode that is unencumbered by coordination with concurrent HTM transactions, to ensure that our system can be competitive with state-of-the-art optimized STMs when HTM is unavailable or ineffective for a given workload.

In this section, we present a simple PhTM prototype we have implemented in order to explore the feasibility of integrating multiple implementations, changing between them, etc., as well as to evaluate the performance improvement possible by eliminating the need for transactions to check for conflicts with concurrent transactions executed using different methods. Our prototype supports only the HARDWARE and SOFTWARE modes mentioned above, with the SOFTWARE mode configured to allow us to "plug in" a variety of STMs for experimentation; to date we have experimented with the STM component of HyTM [3] and a variant on it (see Section 4) and with TL2 [4]. We can include additional modes when and if we are convinced we need them.

3.1 Compiler changes

For prototyping PhTM, we used the HyTM compiler described in [3], with some small modifications to support PhTM. The HyTM compiler generates two code paths for each atomic block. One path executes the block using a hardware transaction and the other executes it using the STM in the HyTM library.

Code generated by the HyTM compiler calls a special HyTM library function before each transaction attempt, and its return value determines whether the transaction is executed using the hardware code path or the software code path. The same functionality is used in PhTM to decide which code path to take, with the library modified to make these decisions based on the current mode, or perhaps to change the mode before proceeding.

Some modifications to the HyTM compiler are necessary in order to support a "bare bones" hardware transaction mode. The first change is needed to allow transactions executed using HTM to ensure that they run only in appropriate modes (HARDWARE mode in our simple example). To support this, the HyTM compiler now supports emitting a call to a new checkMode function in the PhTM library after the beginning of a hardware transaction. If this function sees that we're not in HARDWARE mode, it aborts the hardware transaction (which then retries); otherwise, the transaction proceeds in HARDWARE mode. This allows us to ensure within a hardware transaction that we are in HARDWARE mode; if the mode changes before the transaction completes, it will fail and retry.

As discussed, there is no need in PhTM for a hardware transaction to check for conflicts with software transactions. Therefore, the compiler now supports an option to disable generation of conflict checking calls in the hardware path. We found that these calls accounted for the lion's share of the overhead of HyTM relative to the unbounded LogTM system we compared against, even when we modified the library so that it trivially reports no conflict. This was a driving factor motivating our work on PhTM.

As in HyTM, transactions executed using the software path make calls into the library for beginning and committing software transactions, and for each memory access. Thus we can use the STM we developed for our HyTM system (minus the functions that support conflict checking by hardware transactions), or other STMs that conform to the same interface. To compare against a state-of-the-art STM that is not constrained to interoperate with hardware transactions, we modified the TL2 STM [4] so that it can be plugged in as the STM component of our PhTM prototype. This was relatively straightforward, requiring us to change Tl2's interface to conform to that required by our compiler, and supporting some additional simple functionality, like the ability to self-abort a transaction, and flattening (lexically and dynamically) of nested transactions.

3.2 Implementing PhTM modes

Our simple prototype supports only two modes, which allows a simpler mechanism for coordinating modes than the more general one described in Section 2.

Initially, the system is in HARDWARE mode, and all transactions execute using the hardware path. If we are lucky, all transactions can be executed using HTM. However, because we assume only best-effort HTM, we may encounter a transaction that cannot be committed using HTM, in which case we will need to switch to SOFTWARE mode to complete the transaction. Depending on workload and environment, it will most likely be desirable to return to HARDWARE mode at some

point, so that we can again get the performance benefit of using HTM. In some cases, it may make sense to simply stay in SOFTWARE mode forever if we determine that the best-effort HTM is (almost) never successful. However, our simple prototype always eventually goes back to HARDWARE mode.

The modeIndicator variable used to coordinate mode changes in our simple two-mode prototype has the following structure:

modeIndicator = ⟨ mode, DeferredCount, UndeferredCount⟩

When in HARDWARE mode, DeferredCount represents the number of transactions that must be executed in software after the next mode transition; when in SOFTWARE mode, it represents the number of these transactions that have not yet succeeded. The UndeferredCount represents the number of other transactions currently executing in software (its value is always zero when in HARDWARE mode).

While the system remains in HARDWARE mode, the library always directs every transaction to execute using the hardware path, and the checkMode function reads the global modeIndicator, confirms that the mode is HARDWARE and proceeds with the transaction. When a HTM transaction is unsuccessful for some reason, the library decides whether to retry in hardware or to initiate a transition to SOFTWARE mode before retrying. These decisions may be made based on a variety of factors, including feedback from the HTM about the reason for failure, such as whether it was caused by contention or by exceeding the best-effort HTM's resource constraints. Experience shows that retrying in hardware (after a short delay) makes sense when the failure is due to contention, but not when it is due to exceeding the resources of the best-effort HTM. Our current HTM model does not provide any such feedback, however, so when a transaction's resource needs cause the hardware path to fail, the hardware path fruitlessly retries several times (currently nine) before failing over to SOFTWARE mode. This is obviously a bad situation to be in, and we have seen benchmarks that have a large number of transactions that encounter these resource-need failures perform very poorly. In the future, we plan to enhance our simulated HTM support to give feedback about reasons for failure so we can make better decisions in this regard.

When a transaction decides to initiate a transition to SOFTWARE mode, it attempts to increment DeferredCount while confirming that the mode is still HARDWARE; waits for a short time or for the mode to change to SOFTWARE; and then atomically changes the mode to SOFTWARE (if this has not occurred already) while keeping DeferredCount and UndeferredCount unchanged. When the mode becomes SOFTWARE, the library then directs the transaction to retry using the software path.

Transactions that increment DeferredCount decrement it again when they complete. The system remains in SOFTWARE mode while DeferredCount is nonzero, and during this time, additional transactions can execute in SOFTWARE mode. These transactions first increment UndeferredCount (while confirming that the mode is SOFTWARE and DeferredCount is nonzero); they decrement UndeferredCount when they complete. Transactions that decrement one of the counters to zero while the other is zero simultaneously switch the mode back to HARDWARE. A transaction that attempts to increment UndeferredCount but cannot because DeferredCount is zero simply waits for the mode to become HARDWARE again. Thus, after the deferred transactions all complete, no new undeferred transactions begin, so we return to HARDWARE mode after all undeferred transactions have completed.

4. Performance evaluation

In this section, we present some preliminary performance experiments, designed to evaluate the benefit of being able to execute different kinds of transactions in different phases, rather than requiring the different types of transactions to correctly coexist.

4.1 Experimental platforms

To simulate HTM support, we started with the Wisconsin LogTM simulator [14]. This is a multiprocessor simulator based on Virtutech Simics [12], extended with customized memory models by Wisconsin GEMS [13], and further extended to simulate the unbounded LogTM architecture [14]. We added instruction handlers for the txn_begin, txn_end and txn_abort instructions produced by our compiler, mapping these to the corresponding LogTM instructions. We used this simulator for curves labeled "LogTM" in our graphs, and for simulations not involving HTM.

In LogTM, when a transaction fails, it is retried, transparently to the software (possibly after a short delay to reduce contention). To experiment with best-effort HTM support, we created a variant of the simulator that immediately branches upon failure to a *handler-addr*, specified with the txn_begin instruction. This allows software to decide how to react to failures, as needed by HyTM and PhTM. We further modified the simulator to abort HTM transactions when either (a) the number of store instructions executed exceeds a specified number (32 in the experiments presented here), or (b) a transactional cache line is "spilled" from the L2 cache.

Finally, LogTM has a built-in contention management mechanism that delays memory requests that conflict with a transaction, NACK'ing the processor that issued the request. Memory requests can therefore be repeatedly NACKED for a very long time, until the transaction commits or aborts.[3] This mechanism complicates the hardware design beyond what we would minimally assume about a best effort HTM, for which an incoming conflicting memory request would cause the local transaction to abort. So we further modified the simulator to do just that.

[3] We observed this phenomenon in our early simulations.

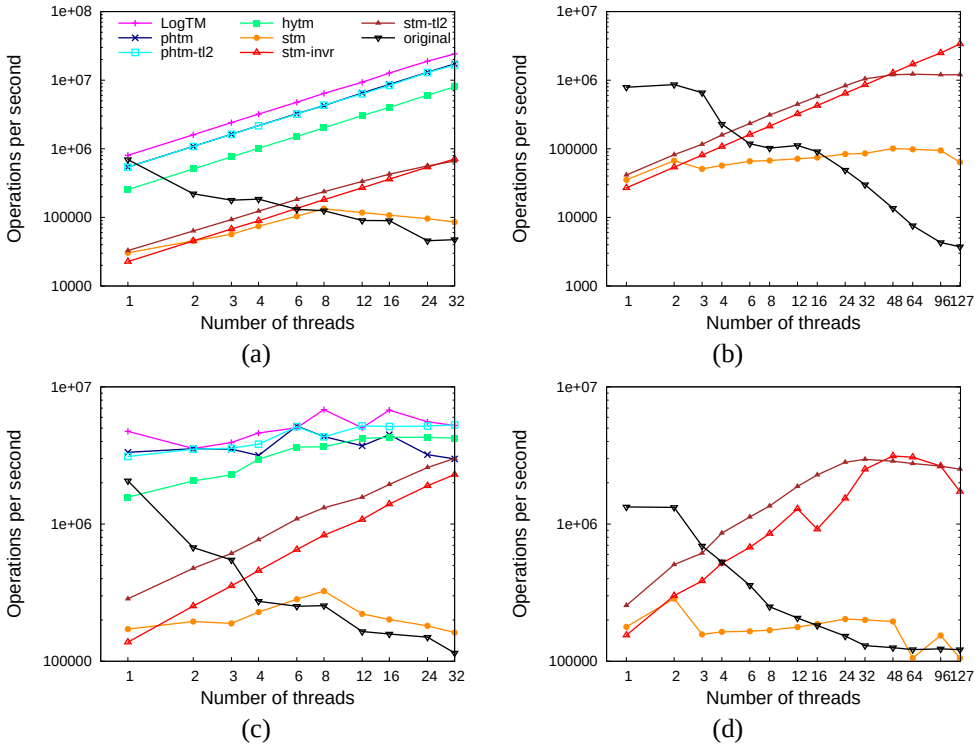

Figure 1. Performance experiments: (a) Berkeley DB on simulator (b) Berkeley DB on E25K (c) Red-Black tree on simulator (d) Red-Black tree on E25K.

Together these changes result in a model that emulates a best-effort HTM design that uses only on-chip caches and store buffers, and aborts transactions that exceed these resources or that encounter any conflict. We used this "neutered" simulator for all PhTM and HyTM simulations.

The systems we simulated share the same multiprocessor architecture described in [14], except that our simulated processors were run at 1.2GHz, not 1GHz, and we used the simpler `MESI_SMP_LogTM` cache coherence protocol, instead of the `MOESI_SMP_LogTM` protocol.

We also conducted experiments not involving HTM support on a Sun Fire™ E25K [17], a shared memory multiprocessor containing 72 1500MHz UltraSPARC™ IV chips (with two hardware threads per chip) and 295 GByte of shared memory, running a 150 MHz system clock. Each processor chip has a 64 KBbye instruction cache and a 128 KByte data cache on chip and a 32 MByte level 2 cache off chip. The processors are organized into nodes containing four chips (eight hardware threads) each.

In all experiments, both simulated and real, we bound each thread to a separate processor to eliminate potential scheduler interactions. Furthermore, while our implementations support various forms of privatization [5, 10], analysis of their impact on performance and scalability is beyond the scope of this work, so we disabled privatization for these experiments.

4.2 Experimental systems

For systems with HTM support, we experimented with LogTM [14], HyTM [3], and the PhTM system described in Section 3 instantiated with the STM component of HyTM (`phtm`) and with TL2 [4] (`phtm-tl2`).

For systems without HTM support, we conducted experiments using TL2, the STM component of HyTM (`stm`), and a variant of it (`stm-invr`) that uses invisible reads instead of semi-visible reads. The semi-visible read mechanism that `stm` uses records how many (but not which) transactions hold read ownership of a location [3]. Using semi-visible reads for HyTM allows hardware transactions to detect conflicts with concurrent software transactions cheaply; and also supports an optimization that often avoids iterating over the read set when validating a transaction [11]. On the other hand, the semi-visible read counters are hotly contended under heavy read-sharing. With the *invisible reads* approach, each transaction keeps track of its reads *privately*, which avoids the contention under heavy read sharing but also requires transactions to iterate over the read set for every validation. TL2 also uses invisible reads, but does not require transactions to iterate over the read set for validation, instead using a global counter to ensure reads are consistent. As we will see, different approaches to read sharing are better for different workloads and machine sizes.

4.3 Experiments with the Berkeley DB lock subsystem

In these experiments, each of N threads repeatedly requests and releases a lock for a different object using the transactified Berkeley DB lock subsystem [3], and we measure the total number of iterations achieved per second. Because different threads lock different objects, there is no inherent requirement for threads to synchronize with each other, so in principle we should observe good scalability.

Figure 1(a) shows results achieved with the simulators described above (axes are log-scale). The curve labelled `original` shows the result when using the production Berkeley DB lock subsystem, which uses one global lock to protect its data structures. As expected, it exhibits very poor scalability, delivering steadily lower throughput with increasing numbers of threads. LogTM consistently delivers the best performance and scales well across the range. For these experiments, the resources of the neutered simulator are sufficient to accommodate all transactions, and there are no conflicts between transactions. As a result, no mode changes occurred in the PhTM systems and no transactions were executed in software for HyTM. Thus, the results show the overhead of being *able* to fall back on alternative methods in case the best-effort HTM is unable to commit a transaction, even when it is not necessary to do so. While HyTM is roughly a factor of three worse than LogTM across the board, `phtm` is only a factor of 1.5 worse. This demonstrates that significant overhead is eliminated by not needing to detect conflicts with concurrent software transactions, which is exactly what motivated us to develop PhTM. As expected when no transactions are executed in software, `phtm-tl2` performed identically to `phtm`. This highlights the flexibility of the PhTM approach as compared to HyTM: the design of the STM used in `SOFTWARE` mode is not constrained by and does not affect the performance of transactions executed in `HARDWARE` mode.

Amongst the STMs, HyTM's `stm` (with semi-visible reads) is outperformed by either TL2 or HyTM's `stm-invr` everywhere, indicating that it is not a good choice for PhTM's `SOFTWARE` mode (though it may be the right choice for a `HYBRID` mode if most transactions succeed using HTM). TL2 outperforms `stm-invr` by about 41% for 1 thread, and continues to outperform it up to 16 threads because it does not need to iterate over read sets to ensure read consistency. However, TL2's global counter begins to impact its performance thereafter, and `stm-invr` scales better, outperforming TL2 by about 7% at 32 threads. Combining the features of these STMs to achieve the best of both would be difficult and messy, but the PhTM approach allows us to switch between different modes, keeping their implementations separate.

We conducted the same experiments on the E25K system mentioned above for cases not requiring HTM support, and observed qualitatively similar results (see Figure 1(b)). TL2 outperformed `stm-invr` by about 35% at 1 and 2 threads, but `stm-invr` outperformed TL2 thereafter. Finally, `stm-invr` scaled well to 127 threads (the largest test we conducted), where it outperformed TL2 by more than a factor of three.

4.4 Experiments with red-black tree

When transactifying the Berkeley DB lock subsystem, we took care to eliminate conflicts between operations locking different objects. Avoiding conflicts is the first priority for scalable performance. However, we cannot ignore performance in the face of conflicts between transactions. For experiments with conflicts, we used a red-black tree data structure. In these experiments, we initialize the tree so that it contains about half of the values from the key range 0..4095, and then measure throughput achieved with each thread repeatedly executing operations on the tree, randomly choosing between insert, delete, and lookup from a 20:20:60 distribution, and a value chosen randomly from 0..4095. (We chose the key range and tree size precisely in order to yield significant contention as we increase the number of threads.) Again we have conducted experiments on the E25K machine and on the simulator.

For the E25K (Figure 1(d)), all STMs show some scalability up to about 16 threads, with `stm` doing worst because of its semi-visible read mechanism, especially because the validation optimization that this mechanism supports is much less effective when there are conflicts. This again demonstrates the advantage of being able to use one of the STMs that is not compatible with HTM transactions when HTM support is not available. Beyond 16 threads, our results are more erratic. We are yet to analyze the reasons for this in detail, but we note that with a fixed key range, contention increases with more threads, so it becomes more challenging to be scalable. We hypothesize that some contention control adjustments may be necessary to achieve more predictable performance. We also note that, while the results are erratic, they do indicate that `tl2` and `stm-invr` are able to maintain throughput, while `stm` predictably drops off dramatically for the reasons described above.

To understand the simulated `rbtree` results (Figure 1(c)), recall that for now we retry the hardware path nine times before resorting to software. For both HyTM and the two PhTM variants, we do see contention, but even for the highest-contention (32-thread) cases, we rarely end up using the software path, with about 0.1% of the HyTM transactions and about 0.1% of the PhTM transactions failing over to software. Furthermore, since all transactions run in software when in `SOFTWARE` mode, a total of about 2% of the PhTM transactions ran in software. Of the software transactions, around 1/3 were aborted and retried due to contention; the rest succeeded (in some cases waiting for contention to abate). In no case did HTM transactions fail due to resource constraints.

Since the fraction of transactions executed using STM is very low, these experiments again demonstrate the cost

of being *able* to fail to software when it is needed. Unlike with the BKDB experiments, though, these experiments do occasionally fail to software, demonstrating that the mode changes do actually happen without having an inordinate effect on performance. Furthermore, because we experience mode changes in these experiments, we can also see the effects of instantiating our PhTM prototype with different STMs.

HyTM again outperforms the best of the STMs across the board. PhTM again provides significant improvement over HyTM. In contrast to the Berkeley DB experiments, `phtm` and `phtm-tl2` do perform differently in these experiments—at 12 or more threads, `phtm`'s performance starts to degrade significantly, doing significantly worse even than HyTM at 32 threads. This may be due to a bad interaction between PhTM's simple mode-changing policy, which we made very little effort to tune, and `phtm`'s STM's current contention-management policy, which may be more aggressive about aborting other software transactions than it ought to be.

4.5 Discussion

It is interesting to ponder what accounts for the remaining gap in performance between PhTM and LogTM, which is a factor of 1.5 for large transactions (Berkeley DB); closer to a factor of two for shorter ones (Red-Black tree). The need to read the `modeIndicator` and check that we are in `HARDWARE` mode imposes some small overhead that is required, but there is also still plenty of room to improve the performance of PhTM. In particular, to keep our compiler and library independent, we have thus far inlined calls to library functions in a way that the compiler does not really understand, making it impossible to for the compiler to perform some basic optimizations around those calls (better register allocation, etc.) that it might otherwise perform. We believe that this and some other simple compilation changes we plan to investigate will eliminate a significant fraction of the remaining gap. (These changes will also improve the performance of all of the other systems we compare against, but we do not expect our conclusions to change as a result.)

Another factor is that LogTM has additional hardware complexity devoted to built-in contention management, while (as discussed above), in our system decisions about how many times to retry are not yet being made as well as we would expect to be able to do given best-effort HTM that provides feedback on the reasons for failure. So we are optimistic that we can further reduce the gap in various ways. Nonetheless, it probably is not possible to eliminate it entirely, and the remaining gap would be the price paid for the benefit of having much simpler hardware.

5. Concluding remarks

We have presented PhTM, which allows us to use different transactional memory implementations at different times in the same application. We have demonstrated that, by eliminating the need for transactions executed in different modes to be compatible with each other, this approach provides a significant performance and flexibility advantage over our previous HyTM implementation. In particular, we can execute transactions using best effort HTM support with performance that is much closer to that of a significantly more complicated unbounded HTM scheme than our previous HyTM prototype. Furthermore, by eliminating constraints on the STM we use when HTM support is unavailable or ineffective for the current workload, we can use state-of-the-art STMs and even switch between different ones for different workloads.

While we have demonstrated the feasibility of switching seamlessly between different modes supporting different transactional memory implementations, we have thus far only experimented with a simple two-mode system, and we have not yet explored strategies for deciding when to switch modes, or to which mode we should switch. Designing a system that behaves close to optimally across a wide range of workloads will be challenging. Nonetheless, based on our results to date, we are confident that we can at least:

- achieve performance very close to what hypothetical unbounded HTM could provide given best effort HTM that is usually effective; and
- be very competitive with state-of-the-art STM-only systems when HTM support is unavailable or ineffective for the current workload.

Thus, we improve on the advantages of our previous HyTM work [3] when HTM support is available, and eliminate the potential disadvantage when it is not.

Future work includes more comprehensive evaluation with a wider range of benchmarks, supporting additional modes, and further improving the performance of our prototype in various modes as well as mechanisms for deciding when to switch to what mode.

Acknowledgments

We are grateful to Sasha Fedorova, Victor Luchangco and Nir Shavit for useful discussions, to Peter Damron for compiler support, and to Brian Whitney for access to the E25K. We are especially grateful to Kevin Moore, for his help in supporting the LogTM simulator and for his guidance in helping us with the significant modifications that we made to the simulator.

References

[1] O. Agesen. GC points in a threaded environment. Technical Report SMLI SMLI TR-98-70, Sun Microsystems Laboratories, Dec. 1998.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Feb. 2005.

[3] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM Press.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. International Symposium on Distributed Computing*, 2006. To appear.

[5] D. Dice and N. Shavit. What really makes transactions faster? In *TRANSACT Workshop*, June 2006. http://research.sun.com/scalable/pubs/TRANSACT2006-TL.pdf.

[6] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable Non-Zero Indicators. In *Proc. 26th Annual ACM Symposium on the Principles of Distributed Computing*, Aug. 2007. To appear.

[7] R. Ennals. Software transactional memory should not be obstruction-free, 2005. http://www.cambridge.intel-research.net/ rennals/notlockfree.pdf.

[8] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. 31st Annual International Symposium on Computer Architecture*, June 2004.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[10] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *ISMM '06: Proceedings of the 2006 international symposium on Memory management*, pages 74–83, New York, NY, USA, 2006. ACM Press.

[11] Y. Lev and M. Moir. Fast read sharing mechanism for software transactional memory, 2004. http://research.sun.com/scalable/pubs/PODC04-Poster.pdf.

[12] P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrom, and B. Werner. SimICS/sun4m: A virtual workstation. In *Proceedings of the USENIX 1998 Annual Technical Conference (USENIX '98)*, June 1998.

[13] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[14] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[15] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005.

[16] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, Special Issue(10):99–116, 1997.

[17] Sun Fire E25K/E20K Systems Overview. Technical Report 817-4136-12, Sun Microsystems, 2005.

[18] M. Tremblay, Q. Jacobson, and S. Chaudhry. Selectively monitoring stores to support transactional program execution. US Patent Application 20040187115, Aug. 2003.