Hybrid Transactional Memory

Peter Damron
Sun Microsystems
peter.damron@sun.com

Alexandra Fedorova
Harvard University and
Sun Microsystems Laboratories
fedorova@eecs.harvard.edu

Yossi Lev
Brown University and
Sun Microsystems Laboratories
levyossi@cs.brown.edu

Victor Luchangco Mark Moir Daniel Nussbaum
Sun Microsystems Laboratories
{victor.luchangco,mark.moir,daniel.nussbaum}@sun.com

Abstract

Transactional memory (TM) promises to substantially reduce the difficulty of writing correct, efficient, and scalable concurrent programs. But "bounded" and "best-effort" hardware TM proposals impose unreasonable constraints on programmers, while more flexible software TM implementations are considered too slow. Proposals for supporting "unbounded" transactions in hardware entail significantly higher complexity and risk than best-effort designs.

We introduce *Hybrid Transactional Memory* (HyTM), an approach to implementing TM in software so that it can use best-effort hardware TM (HTM) to boost performance but does not *depend* on HTM. Thus programmers can develop and test transactional programs in existing systems today, and can enjoy the performance benefits of HTM support when it becomes available.

We describe our prototype HyTM system, comprising a compiler and a library. The compiler allows a transaction to be attempted using best-effort HTM, and retried using the software library if it fails. We have used our prototype to "transactify" part of the Berkeley DB system, as well as several benchmarks. By disabling the optional use of HTM, we can run all of these tests on existing systems. Furthermore, by using a simulated multiprocessor with HTM support, we demonstrate the viability of the HyTM approach: it can provide performance and scalability approaching that of an unbounded HTM implementation, without the need to support all transactions with complicated HTM support.

Categories and Subject Descriptors C.1 [*Computer Systems Organization*]: Processor Architectures; D.1.3 [*Software*]: Concurrent programming

General Terms Algorithms, Design

Keywords Transactional memory

1. Introduction

Transactional memory (TM) [11, 29] supports code sections that are executed *atomically*, i.e., so that they appear to be executed one at a time, with no interleaving between their steps. By allowing programmers to express *what* should be executed atomically, rather than requiring them to specify *how* to achieve such atomicity using locks or other explicit synchronization constructs, TM significantly reduces the difficulty of writing correct concurrent programs. A good TM implementation avoids synchronization between concurrently executed transactional sections unless they actually conflict, whereas the more traditional use of locks defensively serializes sections that *may* conflict, even if they rarely or never do. Thus, TM can significantly improve the performance and scalability of concurrent programs, as well as making them easier to write, understand, and maintain.

Despite significant progress in recent years towards practical and efficient software transactional memory (STM) [1, 5, 8, 9, 10, 29], there is a growing consensus that at least some hardware support for TM is desirable. Herlihy and Moss [11] introduced *hardware transactional memory* (HTM) and showed that bounded-size atomic transactions that are short enough to be completed without context switching could be supported using simple additions to the cache mechanisms of existing processors. Although studies (e.g., [3, 7, 23]) suggest that a modest amount of on-chip resources should be sufficient for all but a tiny fraction of transactions, requiring programmers to be aware of and to avoid the architecture-specific limitations of HTM largely eliminates the software engineering benefits promised by TM. This is a key reason why HTM has not been widely adopted by commercial processor designers.

Recent proposals for "unbounded" HTM [3, 7, 22, 23, 27] aim to overcome the disadvantages of simple bounded HTM designs by allowing transactions to commit even if they exceed on-chip resources and/or run for longer than a thread's scheduling quantum. However, such proposals entail sufficient complexity and risk that we believe they are unlikely to be adopted in mainstream commercial processors in the near future.

We introduce *Hybrid Transactional Memory* (HyTM), a new approach to supporting TM so that it works in existing systems, but can boost performance and scalability using future hardware support. The HyTM approach exploits HTM support *if it is available* to achieve hardware performance for transactions that do not exceed the HTM's limitations, and transparently (except for performance) executes transactions that do in software. Because any transaction can be executed in software using this approach, the HTM need not be able to execute every transaction, regardless of its size and duration, nor support all functionality. HyTM thus allows hardware designers to build *best-effort* HTM, rather than having to take on the risk and complexity of a full-featured, unbounded design.

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.
ACM 1-59593-451-0/06/0010.

To demonstrate the feasibility of the HyTM approach, we built a prototype compiler and STM library. The compiler produces code for executing transactions using HTM, or using the library; which approach to use for trying and retrying a transaction is under software control. A key challenge in designing HyTM is ensuring that conflicts between transactions run using HTM and those that use the software library are detected and resolved properly. Our prototype achieves this by augmenting hardware transactions with code to look up structures maintained by software transactions. Some recent proposals for unbounded HTM maintain similar structures in hardware for transactions that exceed on-chip resources, adding significant hardware machinery to achieve the same goal. In contrast, our HyTM prototype makes minimal assumptions about HTM support, allowing processor designers maximal freedom to design best-effort HTM within their constraints.

The HyTM approach we propose enables the programming revolution that TM has been promising to begin, even before *any* HTM support is available, and to progressively improve the performance of transactional programs as incremental improvements to best-effort HTM support become available. In this way, we can develop experience and evidence to motivate processor designers to include HTM support in their plans and to guide HTM improvements. With the HyTM approach, processor designers are free to exploit useful and clever ideas emerging in recent proposals for unbounded HTM without having to shoulder the responsibility of supporting *all* transactions in hardware. Furthermore, they do not need to foresee and support all functionality that that may be required in the future: additional functionality can be supported in software if its expected use does not justify complicating hardware designs for it. We believe it would be a mistake to forgo the advantages of TM, as well as other important uses for best-effort HTM (see Section 5), until an unbounded design can be found that supports all needed functionality and is sufficiently robust to be included in new commercial processors. We therefore hope our work will encourage hardware designers to begin the journey towards effective support for high-performance transactional programming, rather than delay until they can commit to a full unbounded solution.

In Section 2, we briefly discuss some relevant related work. In Section 3, we describe the HyTM approach and our prototype. Section 4 reports our experience using our prototype to "transactify" part of the Berkeley DB system [25] and some benchmarks. We present preliminary performance experiments in which we use an existing multiprocessor to evaluate our prototype in "software-only" mode, and a simulated multiprocessor to evaluate its ability to exploit HTM if it is available. We conclude in Section 5.

2. Related work

We briefly discuss some relevant related research below; an exhaustive survey is beyond the scope of this paper.

2.1 Bounded and best-effort HTM

The first HTM proposal, due to Herlihy and Moss [11], uses a simple, fixed-size, fully-associative transactional cache and exploits existing cache coherence protocols to enforce atomicity of transactions up to the size of the transactional cache. Larger transactions and transactions that are interrupted by context switches fail.

HTM can also be supported by augmenting existing caches, allowing locations that are read transactionally to be monitored for modifications, and delaying transactional stores until the transaction is complete [32]. Related techniques have been proposed for ensuring atomicity of critical sections without acquiring their locks [26], and for speculating past other synchronization constructs [18]. In these approaches, a transaction can succeed only if it fits in cache. This limitation means that a transaction's ability to commit depends not only on its size, but also on its layout with respect to cache geometry. In addition, it is convenient in many cases to avoid complexity by simply failing the current transaction in response to an event such as a page fault, context switch, etc.

Because such *best-effort* mechanisms do not guarantee to handle every transaction, regardless of its size and duration, they must be used in a way that works even if some transactions fail deterministically. Some proposals [18, 26] address this by falling back to the standard synchronization in the original program, so it is merely a performance issue. Our work brings the same convenience to transactional programs: hardware designers can provide best-effort HTM, but programmers need not be aware of its limitations.

2.2 Unbounded STM

Ananian et al. [3] describe two HTM designs, which they call Large TM (LTM) and Unbounded TM (UTM). LTM extends simple cache-based HTM designs by providing additional hardware support for allowing transactional information to be "overflowed" into memory. While LTM escapes the limitations of on-chip resources of previous best-effort HTM designs (e.g., [11]), transactions it supports are limited by the size of physical memory, and more importantly, cannot survive context switches. UTM supports transactions that can survive context switches, and whose size is limited only by the amount of virtual memory available. However, UTM requires additional hardware support that seems too complicated to be considered for inclusion in commercial processors in the near future.

Rajwar et al. [27] have recently proposed Virtualized TM (VTM), which is similar to UTM in that it can store transactional state information in the executing thread's virtual address space, and thus support unbounded transactions that can survive context switches. Rajwar et al. make an analogy to virtual memory, recognizing that the hopefully rare transactions that need to be overflowed into data structures in memory can be handled at least in part in software, which can reduce the added complexity for hardware to maintain and search these data structures. Nonetheless, designs based on the VTM approach require machinery to support an interface to such software, as well as for checking for conflicts with overflowed transactions, and are thus still considerably more complicated than simple cache-based best-effort HTM designs. The way VTM ensures that transactions interact correctly with other transactions that exceed on-chip resources is very similar to the way HyTM ensures that transactions executed by best-effort HTM interact correctly with transactions executed in software. However, HyTM does not need any special hardware support for this purpose, and thus allows significantly simpler HTM support.

Moore et al. [23] have proposed Thread-level TM (TTM). They propose an interface for supporting TM, and suggest that the required functionality can be implemented in a variety of ways, including by software, hardware, or a judicious combination of the two. They too make the analogy with virtual memory. They describe some novel ways of detecting conflicts based on modifications to either broadcast-based or directory-based cache coherence schemes. More recently, Moore et al. [22] have proposed LogTM, which stores tentative new values "in place", while maintaining logs to facilitate undoing changes made by a transaction in case it aborts. Like the other proposals mentioned above, these approaches to supporting unbounded transactions require additional hardware support that is significantly more complicated than simple cache-based best-effort HTM designs. Furthermore, as presented, LogTM does not allow transactions to survive context switches; modifying it to do so would entail further complexity.

Hammond et al. [7] recently proposed Transactional Coherence and Consistency (TCC) for supporting a form of unbounded HTM. TCC is a more radical approach than those described above, as it fundamentally changes how memory consistency is defined and

implemented, and is thus even less likely to be adopted in the commercial processors of the near future.

All of the proposals discussed above acknowledge various system issues that remain to be resolved. While these issues may not be intractable, they certainly require careful attention and their solutions will only increase the complexity and therefore the risk of supporting unbounded transactions in hardware.

2.3 Hybrid transactional memory

Kumar et al. [12] recently proposed using HTM to optimize the Dynamic Software Transactional Memory (DSTM) of Herlihy et al. [10], and described a specific HTM design to support it. Like us, they recognize that it is not necessary to support unbounded HTM to get the benefits of HTM in the common case. Their motivation is similar to ours, but our work differs in a number of ways. First, our prototype implements a low-level *word-based* TM that can be used in the implementation of system software such as Java™ Virtual Machines, while they aim to optimize an object-based DSTM which requires an existing object infrastructure that is supported by such system software. Our approach therefore has the potential to benefit a much wider range of applications. Furthermore, the approach of Kumar et al. [12] depends on several specific properties of the HTM. For example, it crucially depends on support for nontransactional loads and stores within a transaction. These requirements constrain and complicate HTM designs that can support their approach. Unlike our HyTM prototype, their approach *requires* new hardware support, and therefore cannot be applied in today's systems. Finally, even given HTM support built to their specification, their system cannot commit a long-running transaction because their HTM provides no support for preserving transactions across context switches.

As explained elsewhere [20, 21], simple software techniques can overcome all of these disadvantages, delivering the same benefits as the low-level word-based HyTM approach described here to object-based systems such as DSTM [10]. Lie [15] investigated a similar object-based approach using best-effort HTM as an alternative to using UTM, and concluded from his performance studies that UTM is preferable because it is "not overly complicated". But we believe that UTM *is* too complicated to be included in the commercial multiprocessor designs of the near future, and that Lie's results lend weight to our argument that we can use best-effort HTM to provide better performance and scalability than software-only approaches allow, without committing to a full unbounded HTM implementation.

2.4 Additional HTM functionality

Several groups have recently proposed additional HTM functionality, such as various forms of nesting [4, 24], event handlers and other "escape mechanisms" [19], etc. This work is mostly orthogonal to our own, though as we discuss in Section 5, the HyTM approach gives designers the flexibility to choose not to support such functionality in hardware, or to support it only to a limited degree.

3. Hybrid transactional memory

Transactional memory is a programming interface that allows sections of code to be designated as *transactional*. A transaction either *commits*, in which case it appears to be executed atomically at a *commit point*, or *aborts*, in which case it has no effect on the shared state. A transactional section is attempted using a transaction and if the transaction attempt aborts, it is retried until it commits.

The HyTM prototype described in this paper provides a *word-based* interface, rather than *object-based* one: it does not rely on an underlying object infrastructure or on type safety for pointers. Thus, it is suitable for use in languages such as C or C++, where pointers can be manipulated directly.

The HyTM approach is to provide an STM implementation that does not depend on hardware support beyond what is widely available today, and also to provide the ability to execute transactions *using whatever HTM support is available* in such a way that the two types of transactions can coexist correctly. This approach allows us to develop and test programs using systems today, and then exploit successively better best-effort HTM implementations to improve performance in the future.

The key idea to achieving correct interaction between software transactions (i.e., those executed using the STM library) and hardware transactions (i.e., those executed using HTM support) is to augment hardware transactions with additional code that ensures that the transaction does not commit if it conflicts with an ongoing software transaction. If a conflict with a software transaction is detected, the hardware transaction is aborted, and may be retried, either in software or in hardware.

3.1 Overview of our HyTM prototype

Our prototype implementation consists of a compiler and a library: the compiler produces two code paths for each transaction, one that attempts the transaction using HTM, and another that attempts the transaction in software by invoking calls to the library.

The compiler for our prototype is a modified version of the Sun™ Studio C/C++ compiler. We chose this compiler because we are interested in using transactional memory in future implementations of system software such as operating systems, Java™ virtual machines, garbage collectors, etc. Our proof-of-concept compiler work has been done in the back end of the compiler. As a result, it does not support special syntax for transactions. Instead, programmers delimit transactional sections using calls to special `HYTM_SECTION_BEGIN` and `HYTM_SECTION_END` functions. The compiler intercepts these apparent function calls and translates the code between them to allow it to be executed transactionally, using either HTM or STM.

We assume the following HTM interface:[1] A transaction is started by the `txn_begin` instruction, and ended using the `txn_end` instruction. The `txn_begin` instruction specifies an address to branch to in case the transaction aborts. If the transaction executes to the `txn_end` instruction without aborting, then it appears to have executed atomically, and execution continues past the `txn_end` instruction; otherwise, the transaction has no effect, and execution continues at the address specified by the preceding `txn_begin` instruction. We also assume there is a `txn_abort` instruction, which explicitly causes the transaction to abort.

Because we expect that most transactions will be able to complete in hardware, and of course that transactions committed in hardware will be considerably faster than software transactions, our prototype first attempts each transaction in hardware. If that fails, then it calls a method in our HyTM library that decides between retrying in hardware or in software. This method can also implement contention control policies, such as backing off before retrying. In some cases, it may make sense to retry the transaction in hardware, perhaps after a short delay to reduce contention and improve the chances of committing in hardware; such delays may be effected by simple backoff techniques [2], or by more sophisticated contention control techniques [10, 28]. A transaction that fails repeatedly should be attempted in software, where hardware limitations become irrelevant, and more flexible contention control is possible. Of course, all this should be transparent to the programmer, who need only designate transactional sections, leaving the HyTM system to determine whether/when to try the transaction in hardware, and when to revert to trying in software.

[1] The particular interface is not important; we assume this one merely for concreteness.

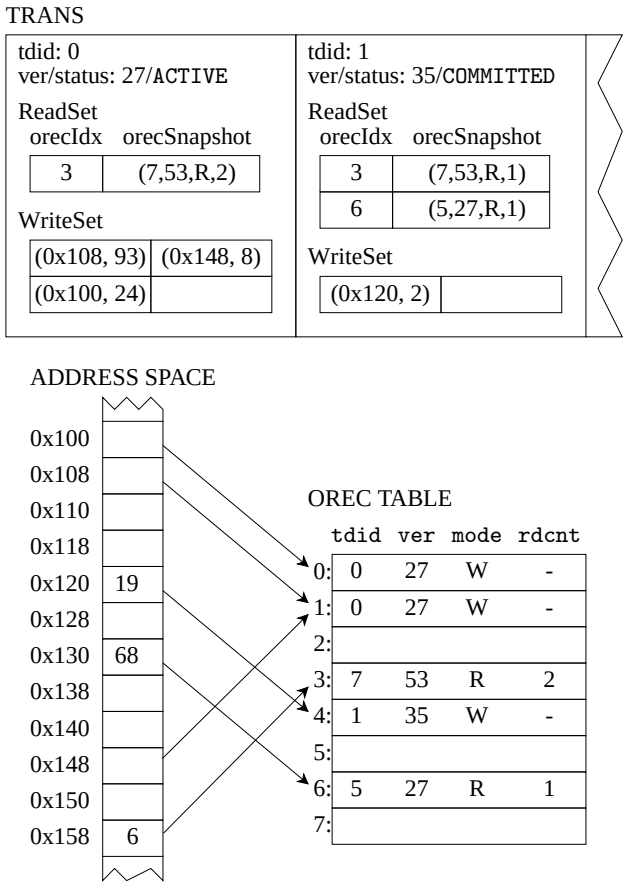


Figure 1. Key data structures for STM component of HyTM.

3.2 HyTM data structures

As in most STMs, software transactions acquire "ownership" for each location they intend to modify. Transactions also acquire "read ownership" for locations that they read but do not modify, but this kind of ownership need not be exclusive. There are two key data structures in our prototype: the *transaction descriptor* and the *ownership record* (orec). Our prototype maintains a transaction descriptor for each thread that may execute a transaction, and a table of orecs.[2] Each location in memory maps to an orec in this table. To keep the orec table a reasonable size, multiple locations map to the same orec. These data structures are illustrated in Figure 1.

A *transaction descriptor* includes a *transaction descriptor identifier* `tdid`, a *transaction header*, a *read set* and a *write set*. The transaction header is a single 64-bit word containing a *version number* and a *status* (which may be `FREE`, `ACTIVE`, `ABORTED`, or `COMMITTED`). The version number distinguishes different (software) transactions by the same thread: a transaction is uniquely identified by its descriptor's identifier and its version number. The read set contains a snapshot of each orec corresponding to a location the transaction has read but not written. The write set contains an entry for each location that the transaction intends to modify, storing the address of the location and the most recent value written to that location by the transaction.

[2] Independently, Harris and Fraser also developed an STM that uses a table of ownership records [8]. Their approach bears some similarity to ours, but the details are quite different. In particular, as far as we know, transactions executed in hardware cannot interoperate correctly with their STM.

An orec is a 64-bit word with `tdid`, `ver`, `mode` and `rdcnt` fields. To avoid interference, orecs and transaction headers are modified using a 64-bit compare-and-swap instruction. The `tdid` and `ver` fields indicate the transaction that most recently acquired the orec in `WRITE` mode. The `mode` field may be `UNOWNED`, `READ`, or `WRITE`, indicating whether the orec is owned, and if so, in what mode. If the orec is owned in `READ` mode, the `rdcnt` field indicates how many transactions are reading locations that map to this orec. This form of read ownership is "semi-visible": a transaction can determine whether any transactions are reading locations that map to this orec—and if so, how many—but it cannot identify the specific transactions doing so.

3.3 Implementing software transactions

A transaction executed using our HyTM library begins with empty read and write sets and its status set to `ACTIVE`. It then executes user code, making calls to our STM library for each memory access. Before writing a location, the transaction acquires exclusive ownership in `WRITE` mode of the orec for that location, and creates an entry in its write set to record the new value written to the location. To acquire an orec in `WRITE` mode, the transaction stores its descriptor identifier and version number in the orec. Subsequent writes to that location find the entry in the write set, and overwrite the value in that entry with the new value to be written.

Similarly, before reading a location, a transaction acquires ownership of the orec for that location, this time in `READ` mode. If the orec is already owned in `READ` mode by some other transaction(s), this transaction can acquire ownership simply by incrementing the `rdcnt` field (keeping all other fields the same). Otherwise, the transaction acquires the orec in `READ` mode, by setting the `mode` field to `READ` and the `rdcnt` field to 1. In either case, the transaction records in its read set the index of the orec in the orec table and a snapshot of the orec's contents at that time.

After every read operation, a transaction *validates* its read set to ensure that the value read is consistent with values previously read by the transaction. (This simple approach is much more conservative than necessary, so there is significant opportunity for improving performance here.) Validating its read set entails determining that none of the locations it read have since changed. This can be achieved by iterating over the read set, comparing each orec owned in `READ` mode to the snapshot recorded previously, ensuring it has not changed (except possibly for the `rdcnt` field). We discuss a way to significantly reduce this overhead in many cases below.

When a transaction completes, it attempts to commit: It validates its read set and, if this succeeds, attempts to atomically change its status from `ACTIVE` to `COMMITTED`. If this succeeds, then the transaction commits successfully. The transaction subsequently copies the values in its write set back to the appropriate memory locations, before releasing ownership of those locations.

The commit point of the transaction is at the beginning of the read validation. The subsequent validation of the reads and the fact that the transaction maintains exclusive ownership of the locations it writes throughout the successful commit implies that the transaction can be viewed as taking effect atomically at this point, even though the values in the write set may not yet have been copied back to the appropriate locations in memory: other transactions are prevented from observing "out of date" values before the copying is performed.

Figure 1 illustrates a state of a HyTM system in which there are 8 orecs and two executing transactions: an active transaction T_0, using transaction descriptor 0 with version number 27; and a committed transaction T_1, using transaction descriptor 1 with version number 35. T_0 has read 6 from address 0x158 (and therefore has a snapshot of orec 3), and has written 93 to 0x108, 8 to 0x148, and 24 to 0x100 (with corresponding entries in its write set). T_1 has

read 6 from 0x158 and 68 from 0x130 (and therefore has snapshots of orecs 3 and 6), and has written 2 to 0x120. Because T_1 has already committed, its writes are considered to have already taken effect. Thus, the logical value of location 0x120 is 2, even though T_1 has not yet copied its write set back (so 0x120 still contains the pre-transaction value of 19). Note that although T_1 is the only executing transaction that has read a location corresponding to orec 6, the transaction descriptor identifier for orec 6 is 5, not 1, because that was the descriptor identifier of the transaction that most recently acquired *write* ownership of that orec.

Resolving conflicts If a transaction T_0 requires ownership of a location that is already owned in `WRITE` mode by another transaction T_1, and T_1's status is `ABORTED`, then T_1 cannot successfully commit, so it is safe for T_0 to "steal" ownership of the location from T_1. If T_1 is `ACTIVE`, this is not safe, as the atomicity of T_1's transaction would be jeopardized if it lost ownership of the location and then committed successfully. In this case, T_0 can choose to abort T_1 (by changing T_1's status from `ACTIVE` to `ABORTED`), thereby making it safe to steal ownership of the location. Alternatively, it may be preferable for T_0 to simply wait a while, giving T_1 a chance to complete. Such decisions are made by a separate *contention manager*, discussed below.

If T_1's status is `COMMITTED`, however, it is *not* safe to steal the orec (because T_1 may not have finished copying back its new values). In this case, in our prototype, T_0 simply waits for T_1 to release ownership of the location.

If T_0 needs to write a location whose orec is in `READ` mode, then T_0 can simply acquire the orec in `WRITE` mode; this will cause the read validation of any other active transactions that have read locations associated with this orec to fail, so there is no risk of violating their atomicity. Again, the transaction consults its contention manager before stealing the orec: it may be preferable to wait briefly, allowing reading transactions to complete.

Read after write If a transaction already has write ownership of an orec it requires for a read, it searches its write set to see if it has already stored to the location being read. If not, the value is read directly from memory and no entry is added to the read set, because the logical value of this location can change only if another transaction acquires write ownership of the orec, which it will do only after aborting the owning transaction. Thus, validation of this read is unnecessary.

Write after read If a transaction writes to a location that maps to an orec that it owns in `READ` mode, then the transaction uses the snapshot previously recorded for this orec to "upgrade" its ownership to `WRITE` mode, while ensuring that it is not owned in `WRITE` mode by any other transaction, and thus that locations that map to this orec are not modified, in the meantime. After successful upgrading, the entry in the read set is discarded, as the orec is no longer owned in `READ` mode.

Fast read validation Our prototype includes an optimization, due to Lev and Moir [13], that avoids iterating over a transaction's read set in order to validate it. The idea is to maintain a counter of the number of times an orec owned in `READ` mode is stolen by a transaction that acquires it in `WRITE` mode. If this counter has not changed since the last validation, then the transaction can conclude that all snapshots in its read set are still valid, so it does not need to check them individually. Otherwise, the transaction resorts to the "slow" validation method described previously.

Nesting Our prototype supports *flattening*, a simple form of nesting in which nested transactions are subsumed by the outermost transaction: it records the nesting depth in the transaction descriptor and ignores `HYTM_SECTION_BEGIN` and `HYTM_SECTION_END` calls for inner transactions so that only outermost transaction commits.

Dynamic memory allocation To ensure that memory allocated during an aborted transaction does not leak, and that memory freed inside a transaction is not recycled until the transaction commits (in case the transaction aborts), we provide special `hytm_malloc` and `hytm_free` functions. To support this mechanism, we augment transaction descriptors with fields to record objects allocated during the transaction (to be freed if it aborts), and objects freed during the transaction (to be freed if it commits).

Contention management Following Herlihy et al. [10], our prototype provides an interface for separable contention managers. The library uses this interface to inform the contention manager of various events, and to ask its advice when faced with decisions such as whether to abort a competing transaction or to wait or abort itself. We have implemented the Polka contention manager [28], and a variant of the Greedy manager [6] that times out to overcome the blocking nature of this manager as originally proposed. We have not experimented extensively with different contention managers or with tuning parameters of those we have implemented.

3.4 Augmenting hardware transactions

We now discuss how our prototype augments hardware transactions to ensure correct interaction with transactions executed using the software library. The key observation is that a location's logical value differs from its physical contents only if a current software transaction has modified that location. Thus, if no such software transaction is in progress, we can apply a transaction directly to the desired locations using HTM. The challenge is in ensuring that we do so only if no conflicting software transaction is in progress.

Our prototype augments HTM transactions to detect conflicts with software transactions at the granularity of orecs. Specifically, the code for a hardware transaction is modified to look up the orec associated with each location accessed to detect conflicting software transactions. The key to the simplicity of the HyTM approach is that the HTM ensures that if this orec changes before the hardware transaction commits, then the hardware transaction will abort.

We illustrate this transformation using pseudocode below. On the left is the "straightforward" translation of a HyTM transactional section, where *handler-addr* is the address of the handler for failed hardware transactions, and `tmp` is a local variable). On the right is the augmented code produced by the HyTM compiler:

```
txn_begin handler-addr      txn_begin handler-addr
                            if (!canHardwareRead(&X))
                              txn_abort;
tmp = X;                    tmp = X;
                            if (!canHardwareWrite(&Y))
                              txn_abort;
Y = tmp + 5;                Y = tmp + 5;
txn_end                     txn_end
```

where `canHardwareRead` and `canHardwareWrite` are functions provided by the HyTM library. They check for conflicting ownership of the relevant orec, and are implemented as follows, where `h` is the hash function used to map locations' addresses to indices into the orec table `OREC_TABLE`:

```
bool canHardwareRead(a) {
  return (OREC_TABLE[h(a)].o_mode != WRITE);
}

bool canHardwareWrite {
  return (OREC_TABLE[h(a)].o_mode == UNOWNED);
}
```

Alternative conflict detection If there are almost never any software transactions, then it may be better to detect conflicts using a single global counter `SW_CNT` of the number of software transactions in progress. Hardware transactions can then just check whether this counter is zero, in which case there are no software transactions with which they might conflict. However, if software transactions are more common, reading this counter will add more overhead (especially because it is less likely to be cached) and will increase the likelihood of hardware transactions aborting due to unrelated software transactions, possibly inhibiting scalability in an otherwise scalable program.

An advantage of the HyTM approach is that the conflict detection mechanism can be changed, even dynamically, according to different expectations or observed behavior for different applications, loads, etc. Thus, for example, a HyTM implementation can support both conflict detection mechanisms described above. Fixing conflict detection methods in hardware does not provide this kind of flexibility. Of course, hardware could provide several "modes", but this would further complicate the designs. Because conflicts involving software transactions are detected in software in HyTM, improvements to the HyTM-compatible STM and methods for checking for conflicts between hardware and software transactions can continue long after the HTM is designed and implemented.

4. Experience and evaluation

In this section, we describe our experience using our prototype to transactify part of the Berkeley DB system, three SPLASH-2 [33] benchmarks (`barnes`, `radiosity`, and `raytrace`), and a microbenchmark `rand-array` we developed to evaluate HyTM.

Because HyTM does not *depend* on HTM support, we can execute all of the benchmarks in existing systems today; we report on some of these experiments in Sections 4.2 and 4.3, and in Section 4.4 we report results of simulations we conducted using the `rand-array` benchmark to evaluate HyTM's ability to exploit HTM support, if available, to boost performance. First, we describe the platforms used for our experiments.

4.1 Experimental platforms, real and simulated

The software-only experiments reported in Sections 4.2 and 4.3 were conducted on a Sun Fire™ 6800 server [31] containing 24 1350MHz UltraSPARC® IV chips [30]. Each UltraSPARC® IV chip has two processor cores, each of which has a 32KB L1 instruction cache and a 64KB L1 data cache on chip. For each processor chip, there is a 16MB L2 cache off chip (8MB per core). The system has 197GB of shared memory, and a 150MHz system clock.

To compare performance of our HyTM prototype with and without various levels of hardware support to an unbounded HTM implementation, as well as to conventional locking techniques, we used several variants of the Wisconsin LogTM transactional memory simulator. This is a multiprocessor simulator based on Virtutech Simics [16], extended with customized memory models by Wisconsin GEMS [17], and further extended to simulate the unbounded LogTM architecture of Moore et al. [22].

Our first variant simply adds instruction decoders and handlers for the `txn_begin`, `txn_end` and `txn_abort` instructions produced by our compiler, mapping these to LogTM's `begin_transaction`, `commit_transaction` and `abort_transaction` instructions. In LogTM, if a transaction fails due to a conflict, it is rolled back and retried, transparently to the software (possibly after a short delay to reduce contention). Thus, there is no need to resort to software transactions, and no need to check for conflicts with them, so we directed the compiler not to insert the usual library calls for such checking in this case. We used this simulator for curves labeled "LogTM" in the graphs presented later, as well as for all experiments not involving HTM, i.e., HyTM in software-only mode, and all conventional lock-based codes.

To experiment with HyTM with HTM support, we also created a variant of the simulator that branches (without delay) to a *handler-addr*, specified with the `txn_begin` instruction, in case the transaction fails. This way, a failed transaction attempt can be retried using a software transaction. This is the simulator used for HTM-assisted HyTM configurations.

Because LogTM is an "unbounded" HTM implementation, transactions fail only due to conflicts with other transactions. To test our claim that the HyTM approach can be effective with "best-effort" HTM, we created another variant of the simulator that aborts HTM transactions when either (a) the number of distinct cache lines stored by the transaction exceeds 16, or (b) a transactional cache line is "spilled" from the cache. This emulates a best-effort HTM design that uses only on-chip caches and store buffers, and fails transactions that do not fit within these resources. We call this the "neutered" HyTM simulator.

The systems we simulated share the same multiprocessor architecture described in [22], 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.

In all experiments, both in real systems and in simulations, we bound each thread to a separate processor to eliminate potential scheduler interactions.

4.2 Berkeley DB lock subsystem

Berkeley DB [25] is a database engine implemented as a library that is linked directly into a user application. Berkeley DB uses locks in its implementation and also exposes an interface for client applications to use these locks. The locks are managed by the *lock subsystem*, which provides `lock_get` and `lock_put` methods. A client calling `lock_get` provides a pointer to the object it wishes to lock. If no client is currently locking the object, the lock subsystem allocates a lock and grants it to the client. Otherwise, a lock already exists for the object, and the lock subsystem either grants the lock or puts the client into the waiting list for the lock, depending on whether the requested lock mode conflicts with the current lock mode.

In the Berkeley DB implementation of the lock subsystem, all data structures are protected by a single low-level lock. The Berkeley DB documentation indicates that the implementors attempted a more fine-grained approach for better scalability, but abandoned it because it was too complicated to be worthwhile. We decided to test the claim that TM enables fine-grained synchronization with the programming complexity of coarse-grained synchronization by "transactifying" the Berkeley DB lock subsystem.

We replaced each critical section protected by the lock with a transactional section. In some cases, a small amount of code restructuring was required to conform with our compiler's requirement that each `HYTM_SECTION_END` lexically matches the corresponding `HYTM_SECTION_BEGIN`. We also replaced calls to `malloc` and `free` with calls to `hytm_malloc` and `htym_free` (see Section 3.3).

We designed an experiment to test the newly transactified sections of Berkeley DB. In this experiment, each of N threads repeatedly requests and releases a lock for a different object. Because they request locks for different objects, there is no inherent requirement for threads to synchronize with each other. The threads do no other work between requesting and releasing the locks.

As expected, the original Berkeley DB implementation did not scale well because of the single global lock for the entire lock subsystem. However, in our initial experiments, the transactified version had similarly poor scalability, and significantly higher cost

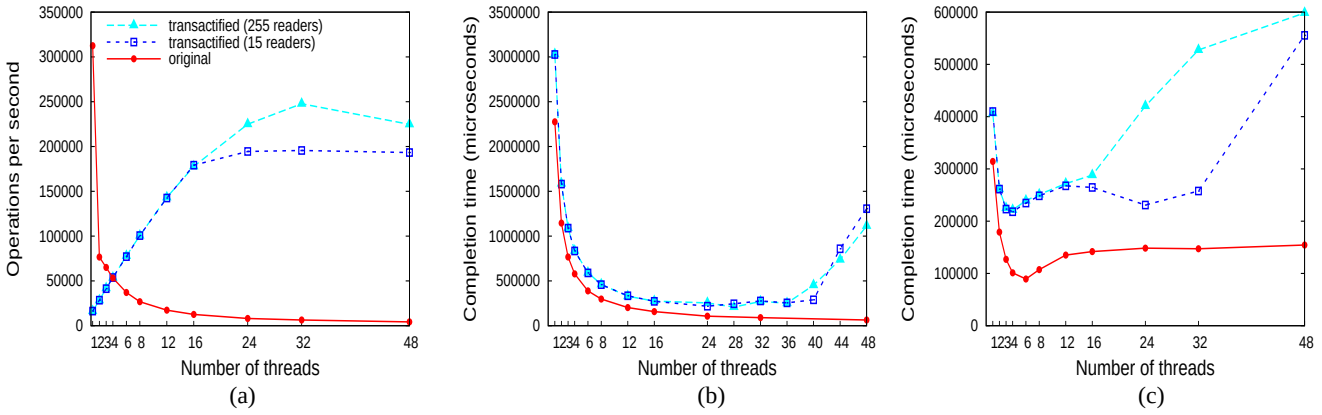


Figure 2. Software-only experiments: (a) Berkeley DB lock subsystem (b) `barnes` (c) `raytrace`

per iteration. We were not surprised by the higher overhead (see Section 4.5), but we were disappointed by the lack of scalability.

A quick investigation revealed that scalability was prevented by false sharing, which occurs when variables accessed by different threads happen to fall in the same cache line, and by two sources of "real" conflict. False sharing can be especially bad in a transactional context because it can introduce unnecessary aborts and retries, which can be much more expensive than the unnecessary cache misses it causes in lock-based programs. Moore et al. [22] make a similar observation from their experience. It is standard practice to "pad" variables in high-performance concurrent programs to avoid the profound impact false sharing can have on performance. We found that this significantly improved the performance and scalability of the transactified version. (Applying these techniques to the original implementation did not improve its scalability, because of the serialization due to the global lock.)

In addition to the conflicts due to false sharing, we found two significant sources of "real" conflicts. First, the Berkeley DB lock subsystem records various statistics in shared variables protected by the global lock. As a result, each pair of transactions conflicted on the statistics variables, eliminating any hope of scalability. It is standard practice to collect such statistics on a per-thread basis and to aggregate them afterwards. However, we simply turned off the statistics gathering (in the original code as well as in the transactified version).

Second, Berkeley DB maintains a data structure for each object being locked, and a "lock descriptor" for each lock it grants. Rather than allocate and free these dynamically, it maintains a pool for each kind of data structure. We discovered many conflicts on these pools because each pool is implemented as a single linked list, resulting in many conflicts at the head of the list. We reduced contention on these pools using standard techniques: Instead of keeping a single list for all the lock descriptors, we distributed the pool into multiple lists, and had threads choose a list by hashing on their thread id. On initialization, we distribute the same number of lock descriptors as in the original single-list pool over the several lists implementing the pool in the revised implementation. We also implemented a simple load-balancing scheme in which, if a thread finds its list empty when attempting to allocate a descriptor, it "steals" some elements from another list. Programming this load balancer was remarkably easy using transactions.

Figure 2(a) compares the original Berkeley DB (with statistics disabled) to two configurations of the transactified version after the modifications described above. For this and other microbenchmarks, we report throughput as operations per second (in this case, a thread acquiring and releasing its lock is one operation); for the SPLASH-2 benchmarks presented later, we report completion time.

When only one thread participated, the transactified version performed roughly a factor of 20 worse than the lock-based version. This is not surprising, as we have thus far avoided a number of optimizations that we expect to considerably reduce overhead of our HyTM implementation, and because with a single thread, the disadvantages of the coarse-grained locking solution are irrelevant. As the number of threads increases, however, the throughput of the original implementation degrades dramatically, as expected with a single lock. In contrast, the transactified version achieves good scalability at least up to 16 threads. For four or more threads, the transactified version beats the lock-based version, despite the high overhead of our unoptimized implementation.

Initially, the `rdcnt` fields in our library had four bits, allowing up to 15 concurrent readers per orec. With this configuration, the transactified version did not scale past 16 threads. A short investigation revealed that the `rdcnt` field on some orecs was saturating, causing some readers to wait until others completed. We increased the number of bits to 8, allowing up to 255 concurrent readers per orec. As Figure 2(a) shows, this allowed the transactified version to scale well up to 32 threads. The decrease in throughput at 48 threads is due to a coincidental hash collision in the Berkeley DB library; changing the hash function eliminated the effect, so this does not indicate a lack of scalability in our HyTM prototype.

4.3 SPLASH-2 benchmarks

We took three SPLASH-2 [33] benchmarks—`barnes`, `radiosity`, and `raytrace`—as transactified by Moore et al. [22], converted them to use our HyTM prototype, and compared the transactified benchmarks to the original, lock-based implementations.

In the original lock-based versions, `barnes` (Figure 2(b)) scaled well up to 48 threads; `radiosity` (not shown) scaled reasonably to 16 threads and thereafter failed to improve performance and even took longer with more threads above 32 threads; and `raytrace` (Figure 2(c)) scaled well only to 6 threads, after which adding threads only hurt performance.

In each case, the transactified version took about 30% longer than the lock-based version with one thread. For `barnes`, the transactified version tracked the original version up to about 24 threads, albeit with noticeable overhead relative to the original version. At higher levels of concurrency, performance degraded significantly. This is because the number of conflicts between transactions increased with more threads participating. We expect to be able to improve performance in this case through improved contention man-

agement: we have not yet experimented extensively with contention management policies, or with tuning those we have implemented.

The original lock-based implementations of `radiosity` and `raytrace` both exhibited worse performance with additional threads at some point: above 24 threads for `radiosity` and above 6 threads for `raytrace`. The transactified versions performed qualitatively similarly to their lock-based counterparts, except that the lack of scalability was more pronounced in the transactional versions, especially for `raytrace`. Again, this is likely due to poor contention management. But it also demonstrates the importance of structuring transactional applications to try to avoid conflicts between transactions. Fortunately, avoiding conflicts in the common case, while maintaining correctness, is substantially easier with transactional programming than with traditional lock-based programming, as illustrated by our experience with Berkeley DB.

Our first transactified version of `raytrace` yielded even worse scalability than shown above. The culprit turned out to be our mechanism for upgrading from `READ` to `WRITE` mode: transactions in `raytrace` increment a global `RayID` variable (therefore reading and then writing it) to acquire unique identifiers. By modifying the benchmark to "trick" HyTM into immediately acquiring the orec associated with the `RayID` counter in `WRITE` mode, we were able to improve scalability considerably. This points to opportunities for improving the upgrade mechanism as well as compiler optimizations that foresee the need to write a location that is being read.

Even after this improvement, `raytrace` does not scale well, largely due to contention for the single `RayID` counter. Recently, a number of research groups (e.g., [4]) have suggested tackling similar problems by incrementing counters such as the `RayID` counter "outside" of the transaction, either by simply incrementing it in a separate transaction, or by using an open-nested transaction. While this is an attractive approach to achieving scalability, it changes the semantics of the program, and thus requires global reasoning about the program to ensure that such transformations are correct. It thus somewhat undermines the software engineering benefits promised by transactional memory.

Note that, with up to 255 concurrent readers per orec, the performance of the transactional version of `raytrace` degraded significantly above 16 threads. This indicates that the original limitation of 15 concurrent readers per orec was acting as a serendipitous contention management mechanism: it caused transactions to wait for a while before acquiring the orec, whereas when all readers could acquire the orec without waiting, the cost of modifying the orec increased because doing so caused a larger number of transactions to abort. This points to an interesting opportunity for contention management, in which read sharing is limited by policy, rather than by implementation constraints. Whether such a technique could be used effectively is unclear.

4.4 `rand-array` benchmark

In this section, we report on our simulation studies, in which we used the simple `rand-array` microbenchmark to evaluate HyTM's ability to mix HTM and STM transactions, and to compare its performance in various configurations against standard lock-based approaches. In the `rand-array` benchmark, we have an array of M counters. Each of N threads performs 1,000 iterations, each of which chooses a set of K counters, and increments all of them in a single transactional section. We implemented three versions: one that uses a single lock to protect the whole array; one that uses one lock per counter; and one that uses a transactional section to perform the increments. To avoid deadlock, the fine-grained locking version sorts the chosen counters before acquiring the locks.

In our initial software-only experiments with the `rand-array` benchmark, even the $K = 1$ case did not scale well for the software-only transactified version. We quickly realized that the address of the array was stored in a global shared variable: because the STM did not know it was a constant, every transaction acquired read ownership of the associated orec, causing poor scalability. We fixed this problem by reading the address of the array into a local variable before beginning the transaction. There are two points to take away from this. First, some simple programming tricks can avoid potential performance pitfalls transactional programmers might encounter. Second, the compiler should optimize STM calls for reading immutable data.

We used the simulators described in Section 4.1 to compare the performance of the `rand-array` benchmark implemented using coarse-grained locking, fine-grained locking, HyTM in software-only mode, HyTM with HTM support, and LogTM. For HyTM with HTM support, we tested two simple schemes for managing conflicts. In the "immediate failover" scheme, any transaction that fails its first HTM attempt immediately switches to software and retries in software until it completes. In the "backoff" scheme, we employ a simple capped exponential backoff scheme, resorting to software only if the transaction fails using HTM 10 times.

Experiments using the neutered simulator (not shown) showed no noticeable difference to the unneutered ones. This is not surprising, as this benchmark consists of small transactions that almost always fit in the cache. The neutered tests will be more meaningful when we experiment with more realistic application codes. Based on studies in the literature [3, 7, 23], we expect that in many cases almost all transactions will not overflow on-chip resources, and thus neutered performance will closely track unneutered performance, even for more realistic codes.

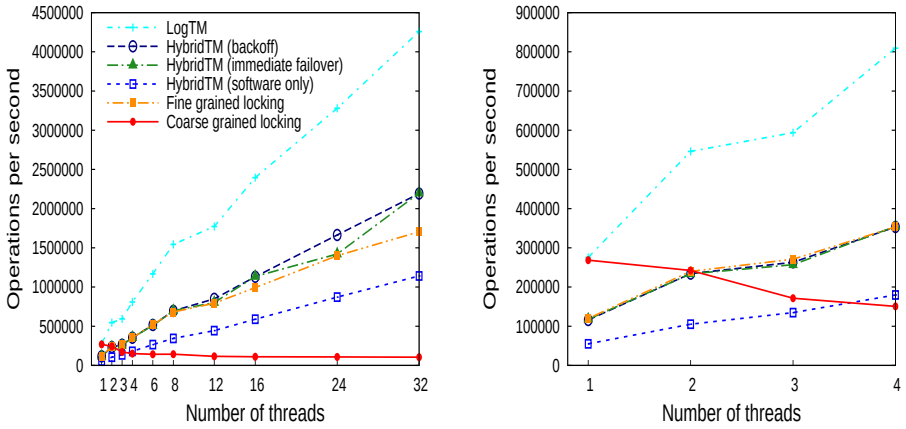
We present simulation data based on the `rand-array` benchmark with $K = 10$; that is, each operation randomly chooses 10 counters out of M and increments each of them. For a "low contention" experiment we chose M=1,000,000 (Figure 3(a)), and for "high contention" we chose M=1,000 (Figure 3(b)). In each experiment, we varied the number of threads between 1 and 32, and each thread performed 1,000 operations. Results are presented in terms of throughput (operations per second). The graphs on the right show a closer look at the graphs on the left for 1 to 4 threads.

First, we observe that with one thread, coarse-grained locking and LogTM provide the highest throughput, while the fine-grained locking and HyTM versions incur a cost without providing any benefit because there is no concurrency available. We explain in Section 4.5 why we expect that HyTM with HTM support can ultimately provide performance similar to that of coarse-grained locking and LogTM in this case.

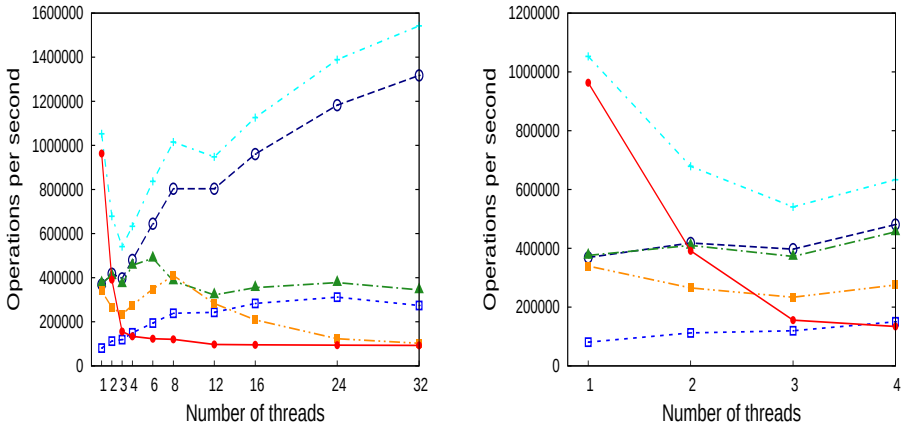
Next, we observe that LogTM provides good scalability in the low-contention experiment (Figure 3(a)), as any respectable HTM solution would in this case (low contention, small transactions). The throughput of the coarse-grained locking implementation degrades with each additional thread, again as expected. Even unoptimized, the software-only HyTM configuration scales well enough to outperform coarse-grained locking for 4 or more threads.

The fine-grained locking approach rewards the additional programming effort as soon as more than 1 thread is present, and consistently improves throughput as more threads are added. The HTM-assisted HyTM configurations provide this benefit without additional programming effort, and outperform even the fine-grained locking implementation for 8 or more threads. In this low-contention experiment, conflicts are rare so the difference between the immediate-failover and backoff configurations is minimal.

Next we turn to the M=1,000 experiment (Figure 3(b)). First, all of the implementations achieve higher throughput for the single-threaded case than they did with M=1,000,000. This is because choosing from 1,000 instead of 1,000,000 counters results in better locality in the 16KB simulated L1 cache. However, due to the increased contention that follows from choosing from a smaller



(a)



(b)

Figure 3. `rand-array` experiments: (a) M=1,000,000 (b) M=1,000 (Closeups on right.)

set of counters, all of them scale worse than they did in the low-contention experiment. Indeed, from 8 threads onwards, adding more threads significantly degrades performance for both coarse-grained *and* fine-grained locking! Meanwhile, the software-only HyTM configuration manages to maintain throughput—though not increasing it much—up to 32 threads. Again, we believe that the lack of scalability is due to poor contention management, and we expect to be able to improve it.

That our unoptimized STM scales better than the hand-crafted fine-grained locking implementation demonstrates the advantages of transactional programming over lock-based programming, even without HTM support. As discussed below, we also see the performance benefits offered by HTM support; with the HyTM approach, we can use simple transactional code in existing systems today, and get the benefit of best-effort HTM support when it becomes available, without changing application code.

LogTM provides the best performance and scalability, as expected. But we also see that the HTM-assisted HyTM configuration with backoff tracks LogTM's scalability well, quickly reducing the performance gap exhibited in the single-threaded case.

We used the immediate-failover HyTM configuration to explore the consequences of the so-called "cascade effect", in which transactions that fail and resort to software then conflict with other transactions causing them too to resort to software, potentially seriously impacting performance. While we do not deny the potential for such a scenario, we believe it can be effectively managed through good contention management policies. This experiment clearly shows the benefit of a simple backoff scheme for HTM transactions, for example. Furthermore, in this high-contention experiment (M=1,000), many transactions conflict, so many transactions resort to software, and therefore scalability is largely determined by that of the software transactions which, as we have already observed, is not yet very good under contention. Our results therefore demonstrate that, even when we force *all* retries to software, and the software contention management is poor, HyTM with best-effort HTM can provide significantly better performance than existing software techniques.

On the surface, it may be surprising that software-only HyTM outperforms the hand-crafted fine-grained locking implementation, as it might be viewed as doing essentially the same thing "under the covers" while adding overhead. However, there is an important difference that is ignored by this simplistic view. When the fine-grained locking implementation encounters a lock that is already held by another thread, it waits, and holds all of the locks it has already acquired. If another thread meanwhile attempts to acquire one of these locks, then it too waits. In this way, long waiting chains can form, essentially serializing the operations involved. As the chains become longer, they become more likely to attract new participants, and eventually we have a "convoy effect", causing the fine-grained implementation to perform little better than the simple coarse-grained one. In contrast, when a software transaction encounters a location that is already held by another transaction,

its contention manager can choose to abort the other transaction and proceed, or to abort itself to avoid impeding other transactions. Even our untuned default contention management policy (adapted from the Polka policy of [28]) is at least somewhat effective in avoiding this effect. We expect that scalability could be improved further by better contention management, and this is an area for further investigation.

There is, of course, a huge space of transactional workloads, ranging over different mixes of transaction sizes, frequency of conflicts between them, mix of reads and writes, etc. We have only scratched the surface, but nonetheless we believe that our experiments demonstrate the viability of the HyTM approach. They also support some useful observations, establish a baseline against which to evaluate future improvements, and point to some directions for future research.

4.5 Discussion

Our main focus to date has been the scalability of HyTM with best-effort HTM support. This has led us to design decisions that compromise on single-threaded performance, as well as on software-only HyTM performance. Ideally, we would like good performance at low contention levels and good scalability, with and without HTM support. Below we discuss some of the tradeoffs, and some of the avenues we think are promising for achieving this goal.

Our use of semi-visible reads (see Section 3.2) requires each software transaction that reads a location to modify its orec in order to allow hardware transactions to detect conflicts in a read-only manner, and also to enable fast validation, as described in Section 3.3. However, if software transactions are frequent (if there is no HTM support, for example), and such reads are frequent, this may impede scalability. A different policy is for HTM transactions (if any) to modify the orecs for locations they modify and to use use "invisible" reads in software transactions, which simply record a snapshot of the orec to be validated later. The tradeoff is that conflicts for locations read this way cannot be detected using the fast validation optimization. These policies can be dynamically mixed; the challenge is in deciding between them.

The performance difference between LogTM and HTM-assisted HyTM in our experiments is due to several factors, some of which can be eliminated by simple techniques like compiler inlining, disabling statistics counters, etc. However, some simple measurements indicate that the difference is dominated by the cost of checking for conflicts with software transactions on each memory access. We see numerous obvious and not-so-obvious optimization opportunities for reducing this overhead.

As explained in Section 3.4, at the risk of impeding scalability when software transactions are frequent, HTM transactions can be made substantially faster by checking a global count of software transactions once *per transaction* (in contrast to per-location checking, or even per-access checking, as in our unoptimized prototype). We plan to try to get the best of both worlds by adaptively choosing between these two conflict detection mechanisms.

To the extent that the performance gap between HTM-assisted HyTM and an unbounded HTM implementation such as LogTM cannot be closed, the remaining difference would be the price paid for the simplification achieved by requiring only best-effort HTM.

We believe that the performance and scalability of software transactions can also be significantly improved through various optimizations and contention management techniques we have not applied to date. Thus, even before HTM support is available, programmers can begin to realize the software engineering and scalability benefits of transactional programming, with the promise of substantial performance gains when HTM support appears.

5. Concluding remarks

We have introduced the Hybrid Transactional Memory (HyTM) approach to implementing transactional memory so that we can execute transactional programs in today's systems, and can take advantage of future "best-effort" hardware transactional memory (HTM) support to boost performance.

We have demonstrated that HyTM in software-only mode can provide much better scalability than simple coarse-grained locking, and is comparable with and often more scalable than even hand-crafted fine-grained locking code, which is considerably more difficult to program. While our prototype would benefit from better contention management and from optimizations that improve single-thread performance, it already performs well enough that transactional applications can be developed and used even before any HTM support is available. Such applications will motivate processor designers to support transactions in hardware, and the fact that HyTM does not require unbounded HTM makes it much easier for them to commit to implementing HTM.

Our work also demonstrates that future best-effort HTM support will significantly boost the performance of transactional programs developed using HyTM today. We hope that this expectation will motivate programmers to consider transactional programming even before HTM is available. HyTM thus creates a synergistic relationship between transactional programs and hardware support for them, eliminating the catch-22 that has prevented widespread adoption of HTM until now, and allowing performance to improve over time with incremental improvements in best-effort HTM support. We therefore believe that the time is right for the revolution in concurrent programming that TM has been promising to begin.

To demonstrate the flexibility of the HyTM approach, we have made minimal assumptions about the functionality and guarantees of HTM support it can exploit. But the HyTM approach is not confined to such simple HTM support. In particular, HTM functionality such as nesting [4, 24], event handlers [19], etc., can be used in HyTM systems, and again, the ability to fall back to software may simplify designs for such features. For one example, hardware might support only a certain number of nesting levels using on-chip resources, and leave it to software to execute deeper nested transactions. The HyTM approach gives maximal flexibility to designers to choose which functionality to support efficiently in hardware and up to what limits, and which cases to leave to software.

Whether unbounded HTM designs will ever be able to provide all functionality required by transactional programs, and whether they will provide sufficient benefit over HyTM implementations to warrant the significant additional complexity they entail is unclear. We encourage designers of future processors to consider whether robust support for unbounded TM is compatible with their level of risk, resources, and other constraints. But if it is not, we hope that our work convinces them to at least provide their best effort, as this will be enormously more valuable than no HTM support at all. Apart from boosting the performance of HyTM, best-effort HTM also supports a number of other useful purposes, such as selectively eliding locks, optimizing nonblocking data structures, and optimizing the Dynamic Software Transactional Memory (DSTM) system of Herlihy et al. [10], as explained elsewhere [20, 21].

Ongoing and future work includes improving the performance and functionality of our prototype, and better integration with languages, debuggers (see [14]), and performance tools.

Acknowledgments

We thank at least the following colleagues from Sun Microsystems for valuable discussions that helped lead to the development of the ideas presented in this paper: Shailender Chaudhry, Bob Cypher, David Detlefs, David Dice, Steve Heller, Maurice Herlihy, Quinn

Jacobson, Paul Loewenstein, Nir Shavit, Bob Sproull, Guy Steele, Marc Tremblay, Mario Wolczko, and Greg Wright. We will be eternally grateful to Kevin Moore for providing his transactified SPLASH-2 benchmarks, and especially for his extensive help with our simulation efforts. We're also indebted to Wayne Mesard and Shesha Sreenivasamurthy for "going the extra mile" in helping us with earlier simulation work that was crucial to the project. We thank Guy Delamarter, Joshua Pincus, Stefan Ebbinghaus, Steve Green, Brian Whitney, and especially Andy Lewis for their generosity with their time and resources in support of our simulation work. Finally, we thank David Wood for useful discussions.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, 2006.

[2] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proc. 16th International Symposium on Computer Architecture*, pages 396–406, May 1989.

[3] 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.

[4] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2006. ACM Press.

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

[6] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proc. 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 258–264, 2005.

[7] 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.

[8] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Oct. 2003.

[9] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25, New York, NY, USA, 2006. ACM Press.

[10] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for supporting dynamic-sized data structures. In *Proc. 22th Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[11] 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.

[12] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2006.

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

[14] Y. Lev and M. Moir. Debuuging with transactional memory. Transact 2006 workshop, June 2006. http://research.sun.com/scalable/pubs/Lev-Moir-Debugging-2006.pdf.

[15] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, May 2004.

[16] 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.

[17] 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.

[18] J. F. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proc. 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, 2002.

[19] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 53–65, Washington, DC, USA, 2006. IEEE Computer Society.

[20] M. Moir. Hybrid hardware/software transactional memory. Slides for Chicago Workshop on Transactional Systems, Apr. 2005. http://www.cs.wisc.edu/~rajwar/tm-workshop/TALKS/moir.pdf.

[21] M. Moir. Hybrid transactional memory, July 2005. http://research.sun.com/scalable/pubs/Moir-Hybrid-2005.pdf.

[22] 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.

[23] K. E. Moore, M. D. Hill, and D. A. Wood. Thread-level transactional memory. *Technical Report: CS-TR-2005-1524, Dept. of Computer Sciences, University of Wisconsin*, Mar. 2005.

[24] M. Moravan, J. Bobba, K. Moore, L. Yen, M. Hill, B. Liblit, M. Swift, and D. Wood. Supporting nested transactional memory in LogTM. In *Proc. 12th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[25] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proc. USENIX Annual Technical Conference*, 1999.

[26] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. 34th International Symposium on Microarchitecture*, pages 294–305, Dec. 2001.

[27] 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.

[28] W. Scherer and M. Scott. Advanced contention management for dynamic software transactional memory. In *Proc. 24th Annual ACM Symposium on Principles of Distributed Computing*, 2005.

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

[30] Sun Microsystems, Inc. http://www.sun.com/processors/ultrasparc-iv/index.xml.

[31] Sun Microsystems, Inc. Sun Fire™6800 Server. http://sunsolve.sun.com/handbook_pub/Systems/SunFire6800/SunFire6800.html.

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

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.