Hybrid Transactional Memory*

Mark Moir
Sun Microsystems Laboratories
1 Network Drive, UBUR02-311
Burlington, MA 01803

July 2005

Abstract

Transactional memory (TM) promises to substantially reduce the difficulty of writing correct, efficient, and scalable concurrent programs. But previous "bounded" and "best-effort" hardware TM implementations impose unreasonable constraints on programmers, while more flexible software TM implementations are considered too slow.

Recent novel proposals suggest supporting "large" and even "unbounded" transactions in hardware. These proposals entail substantially more complexity and risk than best-effort designs. We fear that the perceived need to provide "all or nothing" in hardware will result in designers of mainstream commercial processors opting for "nothing" for the forseeable future.

We describe *Hybrid Transactional Memory*, an approach to implementing TM so that it uses best-effort hardware TM *if it is available* to boost performance, but does not expose programmers to any of its limitatons (or even whether it exists). We also describe several other uses of TM for which a best-effort hardware implementation would provide significant value.

We hope to convince designers of commerical processors that they can provide considerable value with a best-effort hardware TM implementation, without taking on the formidable challenge of supporting *all* transactions in hardware.

1 Introduction

To make reasoning about interactions between threads in a concurrent program tractable, it is often necessary to guarantee that one thread cannot observe partial results of an operation executed by another. In today's software practice, these guarantees are usually provided by using locks that prevent other threads from accessing the data affected by an ongoing operation. Such use of locks gives rise to a number of well known performance and software engineering problems, such as synchronization bottlenecks and deadlock. Efforts to overcome these problems frequently result in complicated and brittle code that is hard to understand, maintain, and improve.

Transactional memory (TM) [9, 19] supports code sections that are executed *atomically*, i.e., so that they appear to be executed one at a time, with no interleaving between the steps of one transaction and another. This allows programmers to write code that accesses and/or modifies multiple memory locations in a single atomic step, significantly reducing the difficulty of writing correct concurrent programs. TM synchronizes concurrent operations only when they conflict, while locks defensively serialize operations that *may* conflict, even if they rarely or never do. Thus, TM can significantly improve performance and scalability of concurrent programs, as well as making them much easier to write, understand, and maintain.

Herlihy and Moss [9] introduced hardware TM (HTM) and showed that atomic transactions up to a fixed size could be supported using simple additions to the cache mechanisms of existing processors, and exploiting existing cache coherence protocols to achieve atomicity.

Rajwar and Goodman [16] have recently proposed Speculative Lock Elision (SLE): hardware support for eliding locks by executing the critical sections they protect atomically without acquiring the lock. They show that this can be achieved with simple additions to existing speculation mechanisms. Martinez and Torrellas [11] have proposed Speculative Synchronization (SS): a technique for speculating past locks, flags, and barriers, again using simple additions to existing speculation mechanisms. These speculation-based mechanisms can be adapted to provide a form of "best-effort" HTM through simple and practical additions to modern commerical processors.

HTM designs based on any of the approaches discussed above cannot support transactions that survive context switches, and furthermore impose architecture-specific limitations on what transactions can be committed because atomicity is achieved using fixed-size on-chip resources. Recent simulation and instrumentation studies (e.g., [2, 4, 15]) suggest that a modest amount of such resources should be sufficient that only a tiny fraction of transactions would exceed these resources. For the SLE technique [16] it does not matter if occasionally a critical section is encountered that exceeds the resources dedicated to lock elision, as the processor can simply acquire the lock in these cases. Similarly, the amount of resources dedicated to the SS technique [11] only affects how far the processor can speculate past a particular synchronization construct.

But for HTM, even if we allocate sufficient on-chip resources that transactions exceed the resources only very rarely, we must still deal with those rare cases correctly: We do not have the luxury of falling back on the lock in the original code or on nonspeculative execution, as the techniques of [16] and [11] do. Requiring programmers to be aware of and to avoid the architecture-specific limitations of a HTM implementation would largely eliminate the software engineering benefits promised by TM. This is a key reason why HTM has not been widely adpoted by commerical processor designers in the dozen or so years since it was originally proposed.

TM can also be implemented in software [5, 7, 14, 19], thus avoiding architecture-specific limitations on transaction size and duration. However, until recently, proposals for software transactional memory (STM) had a number of shortcomings of their own, including poor performance and restrictions and limitations that made them incompatible with standard programming practices. As described in more detail later, we and others have made significant progress in overcoming these limitations and in improving performance so that it is comptetive with lock-based approaches in many cases, and many opportunities for further optimizations remain.

Nonetheless, there is a growing consensus that at least some hardware support for TM is desirable. It is also generally agreed that arbitrary and implementation-dependent restrictions on transaction size and duration are unacceptable. As a result, a number of new HTM proposals have emerged recently that provide hardware support for committing transactions even if they exceed on-chip resources and/or even if they run for longer than a thread's scheduling quantum [2, 15, 17].

We agree with the goals and ideals of the new proposals, but differ on the extent to which complicated hardware support is needed to attain them. In particular, we argue that these ideals can be met even using best-effort HTM, which provides efficient support for as many transactions as resources and design complexity allow, but do not guarantee to be able to commit transactions of any size or duration. To support this position, we describe *Hybrid Transactional Memory* (HyTM), an approach that uses best-effort HTM 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.

The key idea behind HyTM is to augment transactions to be executed in hardware with code to lookup structures maintained by transactions that execute in software to ensure that conflicts

between hardware and software transactions are detected and resolved. Recent proposals for unbounded HTM require significant additional hardware machinery to achieve essentially the same functionality, but as we show, with some simple software techniques, we can exploit standard virtual memory machinery to achieve the same effect. Thus HyTM designs make minimal assumptions about best-effort HTM support, leaving processor designers with maximal freedom in deciding how to support it.

In Section 2, we briefly describe recent proposals for extensions that allow HTM to commit any transaction, without regard to its size or duration. We also point out some issues that illustrate the complexity and constraints involved in making such guarantees, and argue that these may dissuade designers of mainstream commercial processors from attempting to provide them. It is *not* our intention to dismiss these proposals: We encourage processor designers to consider whether they can provide robust support for unbounded HTM based on these or other approaches. And if this is not the case, these proposals nonetheless contain some very useful and clever ideas, which may contribute to successively better best-effort designs over time. The point of this paper is to demonstrate that this journey can start with simple best-effort HTM implementations that can be easily incorporated into the processors of the near future. Whether it is ultimately feasible or desirable to equip processors with HTM support that can commit *any* transaction remains an open question, and in our opinion it would be a mistake to forgo the significant benefits afforded by best-effort HTM until this question is resolved.

In Section 3, we describe the HyTM approach, and attempt an objective discussion of its advantages and disadvantages, as compared to the emerging HTM proposals. We briefly describe several other techniques for which best-effort HTM would be valuable in Section 4. We summarize in Section 5.

2 Recent proposals for unbounded HTM

Ananian et al. [2] describe two HTM designs, which they call Large TM (LTM) and Unbounded TM (UTM). LTM extends simple cache-based HTM designs by allowing transactional information to be "overflowed" into a table allocated in memory by the operating system. If a coherence request is received for a location that may be recorded in the overflow table, the table is searched to determine if a conflict exists. In addition to mechanisms for maintaining and searching this table, hardware and operating system support is required to handle the case in which the table becomes full. While this support does not seem too complicated, every increase in complexity makes a design approach more difficult to integrate with other improvements being considered for the next generation of a commercial processor, and therefore increases the risk that designers will decide not to incorporate HTM support. Furthermore, while LTM escapes the limitations of on-chip resources of previous best-effort HTM designs (e.g., [9]), transactions it supports are limited by the size of physical memory, and 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. This is achieved by allowing transactional information that does not fit in cache to be overflowed into data structures in the executing thread's virtual address space; a transaction can be suspended by migrating in-cache transactional information into these data structures, allowing a transaction to survive even if it is paged out, migrated to a different processor, etc. However, because of the need for hardware to be able to traverse linked data structures, parts of which may be paged out, during execution of a single instruction, UTM seems too complicated to be considered for inclusion in commercial processors in the near future.

Rajwar et al. have recently proposed Virtualized TM (VTM) [17], which is similar to UTM in

that it can store transactional state information in the executing thread's virtual address space, and can 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 cached-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, as we explain later, HyTM does not need any special hardware support to achieve this correct interaction, and indeed there are some advantages to *not* fixing such support in hardware.

Moore at al. [15] 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. Like the other proposals mentioned above, their approach to supporting unbounded transactions requires additional hardware support that is significantly more complicated than simple cache-based best-effort HTM designs.

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 almost certainly increase the complexity and therefore the risk of supporting unbounded transactions in hardware.

Hammond et al. [4] recently proposed Transactional Coherence and Consistency (TCC) for supporting a form of unbounded HTM. TCC represents a much more radical approach than those described above, as it fundamentally changes how memory consistency is defined and implemented. Regardless of its merits, we consider that TCC is too radical to be considered by designers of the commerical processors of the near future, and we do not consider it further.

3 Hybrid transactional memory

With the HyTM approach, any transaction can be attempted either using best-effort HTM (if available) or in software. In the following sections, we describe how this can be achieved in practice. We first briefly describe how—transparently to the programmer—a transaction can be tried in hardware, and then if it does not succeed immediately, retried in hardware and/or software. Next we present a simple HyTM-compatible STM design, and then explain how hardware transactions can be made to interoperate correctly with it, without any special hardware support.

3.1 Integrating hardware and software transactions

As discussed earlier, we expect that almost all transactions will be able to complete in hardware, and of course we expect that small transactions committed in hardware will be considerably faster than software transactions. Therefore, it generally makes sense to try a transaction first in hardware. Transactions attempted in hardware may fail to commit for a variety of reasons, including exceeding limitations on hardware resources or functionality, contention with other transactions, and the occurrence of events across which the particular best-effort HTM cannot sustain transactions (e.g. preemption, page faults, interrupts, etc.).

In some cases, it will 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 [1], or by more sophisticated contention control techniques [7, 18]. Some transactions are doomed to never succeed in hardware, so eventually a transaction that fails repeatedly should be attempted in software, where hardware limitations become irrelevant, and more flexibility for contention control is available.

Of course, all of this should be transparent to the programmer, who should write transaction code once using some convenient language notation, leaving the HyTM system to determine whether/when to try the transaction in hardware, and when to revert to trying in software. This is achievable by having the compiler produce at least two code paths for each transaction, one that attempts the transaction using HTM, and another that attempts the transaction by invoking calls to a HyTM-compatible STM library. The compiler can also produce "glue" code that handles retrying in hardware, reverting to software, etc. It can hardcode trying the transaction at least once in hardware, as we expect this to succeed in the common case, and can insert calls to HyTM library functions that implement contention control and guide decisions about whether to retry in hardware or in software. We also note that if dynamic compilation is used, the code can adapt to observations about transactions during execution. For example, if a particular transaction never succeeds in hardware because it always exceeds the hardware's limitations, then the compiler can remove the hardware attempt—which always wastes time in such cases—from the common case.

A best-effort HTM can provide valuable information regarding the reasons for transaction failure that can guide decisions about whether to retry in hardware or software, and what contention control techniques to employ. But again, the HyTM approach makes minimal assumptions about best-effort HTM, and can be effective even in the absence of such functionality.

3.2 HyTM-compatible STM

HyTM implementations must work regardless of the limitations of any available HTM, or even if no HTM support is available at all. Therefore, a HyTM implementation minimally comprises a complete STM implementation. All STMs in the literature before our work[1] suffer from one and in most cases several of the following serious shortcomings, which preclude their use in practical transactional programming systems:

- the need to statically declare the set of all "transactable" locations before any transactions are executed;
- the need to specify all locations to be accessed by a transaction at the beginning of a transaction;
- space overhead per transactable location;
- unacceptable restrictions on the size of data that can be stored in a transactable location;
- unrealistic assumptions about existing hardware support for synchronization;
- complicated and expensive.

[1]The STM presented by Harris and Fraser [5] overcomes all of these shortcomings, using techniques that bear some similarity to ours, though the details are different. In particular, as far as we know, transactions executed in hardware cannot interoperate correctly with their STM. Harris and Fraser's work was done concurrently with and independently of ours.

Below we demonstrate that it is possible to build an STM that overcomes these shortcomings *and* enables transactions executed in hardware to coexist correctly with those executed by the STM. While we describe some features that could be offered by hardware to support particularly simple and efficient HyTM-compatible STM implementations, we also insist that at least some HyTM implementations based on our approach are applicable in existing systems. This way the HyTM approach allows programmers to develop and test code that uses TM now, and improved STM implementations and/or HTM becoming available in the future can be immediately translated into performance benefits.

3.3 A simple HyTM-compatible STM

Almost all STM designs can be described at a high level as follows. In order to effect changes to multiple memory locations atomically, a transaction acquires exclusive "ownership" of each location it intends to modify, and maintains a *writeset*, which records the new values that will be written to these locations when and if the transaction successfully commits. Each transaction's status is usually recorded in a single word that can be atomically changed, for example from `ACTIVE` to `COMMITTED` or `ABORTED`.

When the transaction changes its status from `ACTIVE` to `COMMITTED`, the new values in its writeset are deemed to be "logically" stored to their respective locations, even though they may not be physically stored in those locations at that time. Typically, the new values are subsequently transferred to the physical locations and then the transaction releases ownership of those locations, thereby allowing other transactions to acquire ownership of them. To ensure atomicity of a transaction that reads some locations without writing them, some STM designs also require transactions to acquire ownership of those locations. However, this ownership need not be exclusive: it is acceptable for another transaction that is reading but not writing such a location to simultaneously hold "shared" ownership of the location.

Many of the shortcomings of previous STMs derive from the way ownership is represented and maintained. In particular, many such STMs co-locate ownership information together with program data, resulting in per-location space overhead and impacting the way in which data is laid out by the compiler. In many cases, the ownership information must be stored together with program data in the same machine word, resulting in restrictions on program data values that are unacceptable in practice. To avoid these problems, we need to store information about which transactions own which locations in a separate structure whose size does not depend on the total number of transactable locations in the system, and whose structure does not depend on predeclaring the set of all transactable locations. For reasonable performance, it must be possible to look up ownership information (if any) for a given location quickly. All of this suggests some form of hash table for representing ownership.

To keep our presentation simple, the basic HyTM-compatible STM design we describe in this section uses a very simple hashing scheme based on a table of *ownership records* (orecs). The orec used for a given memory location is determined by a hash function on the location's address. By choosing the size of the ownership table to be a power of two, we can implement a simple hash function very efficiently by using selected bits of the address to index the table. Note that multiple locations can map to the same orec. With care, this does not affect correctness. However, it may affect performance if too many "false conflicts" occur (i.e., two transactions appear to conflict because they access different locations that hash to the same orec). If so, this problem may be addressed by a larger ownership table, or by a more sophisticated hashing mechanism that avoids false conflicts.

We now describe a very simple HyTM-compatible STM design based on this structure. The sim-

ple design described below uses the compare-and-swap (CAS) instruction to effect atomic changes to orecs and transaction headers. The CAS instruction accepts the address of a memory location, an old value, and a new value, and atomically stores the new value to the to the location if the location currently contains the old value, and leaves memory unchanged otherwise, giving a failure indication. It is straightforward to adapt our design to instead use the load-linked/store-conditional (LL/SC) synchronization instructions [13], and thus HyTM designs based on our description are applicable in all modern shared-memory multiprocessors.

In our simple STM design, each orec consists of the following fields: `o_tid`, `o_version`, and `o_mode`. The `mode` field contains `UNOWNED`, `READ`, and `WRITE`, indicating whether the orec is owned, and if so, in what mode. If the orec is owned, then the `o_tid` field indicates which transaction owns it. The `o_version` field is a "version number", used to avoid the well-known ABA problem [12].

Each transaction record consists of a *transaction header*, which contains a `t_status` field and a `t_version` field, and a *readset* and a *writeset*. The `t_status` field contains `FREE`, `ACTIVE`, `ABORTED`, or `COMMITTED`.

When a transaction begins, it changes a `FREE` transaction to `ACTIVE`, and initializes the transaction's read and writesets. As the transaction executes, loads and stores acquire ownership of the corresponding orecs in the appropriate mode and maintain the read and write sets. Loads first determine whether the transaction has previously stored to the target location, in which case the most recently stored value is returned from the write set. Otherwise, the load acquires the corresponding orec in (at least) `READ` mode, and returns the value from the target location. A store updates the value in the write set if the target location has been stored by the transaction previously, and otherwise ensures that the transaction owns the corresponding orec in `WRITE` mode, and creates an entry with the address of the target location and the new value in the transaction's writeset.

A transaction commits by using CAS to change its status from `ACTIVE` to `COMMITTED`. If this is successful, the transaction then copies all of the values in its writeset to the respective target locations, and then releases ownership of all orecs it acquired.

If a transaction requires ownership of an orec that is already owned by another transaction, several possibilities arise. First, if the "enemy" transaction is aborted, it is safe to "steal" the orec from it. If the enemy transaction is active, then it is not safe to immediately steal the orec, because the enemy transaction might commit, and the loss of the orec would jeopardize its atomicity. Therefore, a decision must be made in this case whether to abort the transaction so we can steal its orec, or to give it more time to complete. Following Herlihy et al. [6, 7], this policy decision is made by a separate contention manager. If the enemy transaction is already committed, in the simple HyTM-compatible STM presented here, we simply wait for the enemy to complete its copyback and release phase. Because of this, this simple design does not provide a nonblocking STM implementation, and thus may be susceptible to performance disadvantages of using locks for synchronization.

Note, however, that a) the software engineering problems associated with using locks are avoided, because the programmer is not aware of the locks, and b) because the blocking part of the implementation is short and is entirely under control of the STM implementation, various techniques can be used to mitigate the performance disadvantages of locking. For example, some operating systems allow a thread to make a library call that discourages the scheduler from preempting the thread for a short while.

Nonetheless, a nonblocking copyback mechanism is generally preferable, and this can be achieved, for example using techniques similar to the one used by Harris and Fraser's STM [5]. Such techniques are quite complicated and the price we pay in the common case to avoid the ill effects of blocking is significant; we are currently working on eliminating this problem. There are also

a variety of pragmatic "mostly nonblocking" schemes, for example that exploit operating system scheduler hooks so that if a thread is descheduled while executing a copyback, another thread can take over this responsibility rather than waiting for the first thread to be rescheduled.

Approaches for achieving nonblocking copybaack are mostly beyond the scope of this paper, as the simple blocking design suffices to explain how an STM can be made compatible with transactions executed by HTM. However, one important note is that best-effort HTM can substantially simplify the task of designing an efficient and nonblocking copyback mechanism by making some simple guarantees. The main difficulty in designing such mechanisms is that, if a thread is delayed while copying back its writeset, and another thread "helps" it in some way in order to proceed with its own transaction without waiting for the delayed thread, then the delayed thread may incorrectly copy back an out-of-date value when it resumes execution. A best-effort HTM implementation that guarantees to (eventually) be able to commit any transaction that reads one cache line and writes one cache line solves this problem trivially because we can use such transactions to ensure that the transaction's copyback phase is still in progress when it performs each copyback store, so the late store problem is eliminated.

3.4 Correct interaction between hardware and software transactions

The key idea behind HyTM is to augment transactions to be attempted in hardware with additional code that ensures that the transaction does not commit if a conflict with an ongoing software transaction is possible. A wide variety of techniques are possible to achieve this, and we describe a couple of them here to demonstrate that these techniques can be simple, efficient, and practical.

For convenience, we assume an interface for hardware transactions, though the interface chosen is not particularly important to the ideas we present. We assume that a transaction is started by the `txn_begin` instruction, which specifies an address to branch to in case the transaction aborts, and that the transaction is ended using the `txn_end` instruction. We further assume a `txn_abort` instruction, which explicitly causes the transaction to abort. If the transaction executes to the `txn_end` instruction without encountering contention or other conditions that prevent it from committing, 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.

Observe from the description of the software transactions in the previous section that a location's logical value differs from its physical contents only because of a current software transaction that modifies that location. Thus, if we can be sure that no such software transaction is in progress, we can apply a transaction directly to the desired locations using HTM. The question is how to ensure that we do so only if no conflicting software transaction is in progress during the hardware transaction.

One very simple way is to ensure that *no* software transaction is in progress. This can be achived by having software transactions increment a global counter `SW_CNT` before beginning and decrement it after completing, and to modify transactions to be executed in hardware so that they check that the counter is zero, and abort if not. For example, transactional code like that on the left below would be transformed to code like that on the right:

```
txn_begin handler-addr
transaction code
txn_end
```

```
txn_begin handler-addr
if (SW_CNT > 0) then txn_abort;
transaction code
txn_end
```

The key to the simplicity of the HyTM approach is that the HTM ensures that the condition checked by the transformed code remains true throughout a transaction that is successfully committed. Observe that, even if a software transaction begins *after* an ongoing hardware transaction checks `SW_CNT`, this will cause the hardware transaction to abort unless it has already committed.

In scenarios in which virtually all transactions succeed in hardware, `SW_CNT` will usually be zero, and will usually be cached, so the overhead of the additional checking required for compatibility with software transactions will be very low. However, with this simple conflict detection scheme, even the occasional transaction that retries in software for whatever reason will cause *all* other hardware transactions to abort and retry, and possibly some of them will retry in software, thereby perpetuating the situation. For a HyTM implementation to be effective across a range of workloads, it should avoid this by also providing a more fine-grained approach to conflict detection.

One way to support finer-grained conflict detection is to further augment code for hardware transactions to detect conflicts at the granularity of orecs. Specifically, the code for hardware transactions can lookup the orec associated with each location accessed to detect conflicting software transactions. Because software transactions indicate their mode of ownership (`READ` or `WRITE`), a hardware transaction does not have to abort if it reads a location in common with a software transactions that is also reading it. (The simple HyTM-compatible STM presented above does not support read sharing between multiple software transactions, but it is not hard to do so.) Thus, for example, the transaction code on the left might be translated to the code on the right (`tmp` is a local variable):

```
txn_begin handler-addr
tmp = X;
Y = tmp + 5;
txn_end
```

```
txn_begin handler-addr
hybridtm_read_check(&X);
tmp = X;
hybridtm_write_check(&Y);
Y = tmp + 5;
txn_end
```

where `hybridtm_read_check` and `hybridtm_write_check` are functions provided by the HyTM library, which check for conflicting ownership of the relevant orec. If `h` is the hash function used to map locations' addresses to indexes into the orec table `OWNER`, then these functions might be implemented as follows:

```
hybridtm_read_check(a) {
   if (OWNER[h(a)].o_mode == WRITE)
      txn_abort;
}
```

```
hybridtm_write_check(a) {
   if (OWNER[h(a)].o_mode != UNOWNED)
      txn_abort;
}
```

A strength of the HyTM approach is that the conflict detection mechanism can be changed, even dynamically, according to different expectations or observed behaviour for different applications, loads, etc. For example, if almost all transactions succeed in hardware, then we can execute streamlined code that only checks `SW_CNT` once at the beginning of the transaction. However, when software transactions are more common, reading this counter will add more overhead (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. In these circumstances, it may be preferable to avoid reading `SW_CNT` and only check the orec table for conflicts.

Fixing conflict detection methods in hardware, as suggested in [17], does not provide this kind of flexibility. Of course, such designs could be refined to provide several different "modes", but this would further complicate the designs, especially because the designs would then need either reliable

machinery for switching between modes automatically, or an interface to software for switching modes. Because conflict detection is done in software in the HyTM approach, improvements to the HyTM-compatible STM and methods for checking for conflicts between hardware and software transactions can continue long after the best-effort HTM is designed and implemented.

Recall that we chose the structure of the orec table and the mapping from location addresses to orecs such that we can compute the address of the associated orec from a location's address very quickly. Furthermore, program locality will translate to orec locality: If the location to be accessed is cached, it is likely that so too is the associated orec. And because the load of the orec and the load of the location are independent loads (we do not need to wait for the data in the location to determine where its orec resides), on a cache miss for the location, the processor can fetch the location and its orec from memory in parallel. Thus, the performance penalty for making hardware transactions interoperate correctly with software ones should be very low.

As presented above, the finer-grained conflict detection mechanism incurs overhead for a call to a library function for every load or store. This keeps our presentation simple, and in fact the experimental prototype compiler developed for us by Peter Damron follows this approach, which allows us to experiment with different methods without changing the compiler. A highly optimized implementation would arrange for common-case checking to avoid the overhead of a library call, for example using inlining or dynamic compilation.

3.5 Advantages and disadvantages of the HyTM approach

The HyTM approach to supporting transactional programming has at least the following advantages over proposals that require special harware support for all transactions:

- Best-effort HTM provides value for those transactions that it can support using whatever level of hardware resources, complexity, and risk that a processor's designers can tolerate. We believe that relieving designers of the need to support *all* transactions will significantly simplify their task, and improve the chances of hardware support for transactions appearing in mainstream commercial processors.

- HyTM can be implemented entirely in user-level code, with no special operating system support.

- A wide variety of conflict detection and contention management methods can be used and interchanged, even dynamically, in response to different applications, loads, etc.

- By supporting transactional programming while depending on minimal assumptions about hardware support, a great deal of flexibility and scope for improvement exists, even after HTM designs are finalized and implemented; unbounded HTM proposals such as [2, 17] fix in hardware at least some aspects of the interaction between on-chip and off-chip transactional information, and therefore heavily constrain future improvements without further hardware modifications.

- HyTM is *extensible*: additional functionality can be supported by software transactions, even if it is not foreseen or is too difficult or not worthwhile to support in hardware. Some possibilities include:

  - early release [7]
  - support for certain kinds of I/O in transactions through software "compensation actions" that "undo" the effect of the I/O if a transaction is aborted

- transactions that have side effects even if they abort, for example exceptions, logging, statistics gathering
- forms of transaction nesting that are more flexible than the simple transaction flattening considered by most TM designs to date [2, 7, 17]
- integration with debugging tools

Of course, if transactions use such functionality excessively, that will diminish the performance advantages provided by the best-effort HTM, but we can continue to exploit these advantages for transactions that do not use them.

- No special support is required when a thread executing a transaction is executed or suspended: Hardware transactions can simply be aborted, and state for software transactions is already entirely stored in the address space of the executing thread, and is therefore managed by standard virtual memory mechanisms. In contrast, recent proposals [2, 15, 17] either cannot sustain transactions across such events, or require special support to "spill" transactional state into memory. This threatens to substantially increase worst-case context switch time, which may be unacceptable, for example when a real-time thread preempts a non-real-time thread that is executing a transaction.

Naturally, HyTM also has some potential disadvantages. One is that, because HyTM-compatible STMs allow the logical value of a memory location to differ temporarily from its physical contents, an issue arises if a memory location is accessed with ordinary load and store operations concurrently with software transactions that access them. While we can usually specify the guarantees made in such cases for a given STM implementation, different implementations make different guarantees, so mandating particular behavior is undesirable. In contrast, proposals for supporting all transactions in hardware can cleanly integrate interactions between transactions and ordinary load and store instructions.

However, it is straightforward to conservatively translate all ordinary load and store instructions that might access memory locations concurrently with new transactional code into mini-transactions to ensure correct interaction; this can even be avhieved with binary translation if source code is not available. There are a host of obvious and not-so-obvious optimizations to avoid translating ordinary loads and stores into transactions, or to mitigate the cost of doing so. For code written in future transactional programming languages, standard typing techniques can be used to distinguish transactional and nontransactional variables, and to ensure correct access to each kind without imposing the overhead of translating ordinary loads and stores into transactions.

The HyTM approach applied using a static compiler results in some code growth due to the need to produce separate code paths for trying a transaction in hardware and in software. If a particular transaction sometimes succeeds in hardware and sometimes in software, this disadvantage may apply to some extent even when using dynamic compilation techniques. If this problem were considered serious enough, a particular best-effort HTM implementation might address it by supporting modes for executing the same code either as a hardware transaction or as a software transaction that calls appropriate HyTM library routines for loads and stores. While this would complicate a particular best-effort HTM design, this is not a requirement of the HyTM approach in general, so does not constrain other processor designers.

Finally, each HyTM transaction is committed entirely using HTM or entirely using STM. Thus, if a transaction slightly exceeds the on-chip resources dedicated to best-effort HTM, it will have to be retried in software in its entirety, while recent unbounded HTM proposals [2, 15, 17] can continue execution of the transaction without aborting. However, given that we expect almost all

transactions to succeed in hardware, this cost applies to a tiny fraction of transactions. Thus, it is not clear that the additional hardware complexity needed to avoid this is warranted.

4 Other uses for best-effort HTM

Transactional lock avoidance Speculative Lock Elision [16] has the potential to provide significant performance improvements for existing lock-based code, because it allows critical sections that do not conflict to execute in parallel. However, it also has the potential to *harm* performance in some cases, because, in a variety of circumstances that the hardware cannot predict, the speculation must be abandoned and the lock acquired as usual.

By using a software technique similar to the key technique in HyTM, we can use best-effort HTM *in selected cases* to execute critical sections without holding their locks. Within a transaction, we read the lock, check that it is not held (aborting if it is), and then execute the critical section. If another thread acquires the lock while we are executing the critical section, the transaction fails, so no harm is done and we can retry or acquire the lock. Similarly, if the critical section exceeds the limitations of the best-effort HTM, we always have the option to acquire the lock.

By applying this technique *selectively* (either manually, or with compiler or library support), we can avoid cases in which it is likely to harm performance. Furthermore we avoid the specialized hardware machinery of SLE that is useful only for the purpose of avoiding locks, instead using best-effort HTM support that is useful for a variety of purposes.

Optimizing lock-free data structures A well known technique for implementing lock-free data structures [8] is to maintain *copies* of the data structure and a pointer to the *current* copy. To apply an operation, a thread copies the current copy to a private copy, applies the operation to the private copy using sequential code, and then attempts to atomically modify the current pointer to make its private copy become the current copy, retrying if a concurrent operation of another thread modifies the pointer first. While conceptually very simple, this approach suffers from two serious drawbacks: a) it does not permit multiple successful operations to apply their operations in parallel, even if they access disjoint parts of the data structure, and b) every operation must copy the entire data structure at least once.

Another variation on the HyTM technique eliminates both problems in the case in which the operation can be completed using a hardware transaction supported by best-effort HTM. The transaction reads the current pointer, and an indication of whether any operations are using the normal method, and if not proceeds to execute the operation in a transaction. If it succeeds, then the operation is completed *without* copying the data structure, and because the transaction only reads (and does not write) the current pointer and normal-case indicator, multiple such operations can succeed in parallel. The normal-case indicator is required because, without it, an operation proceeding in the original manner may take an inconsistent copy of the data structure due to concurrent successful transactional operations.

Optimizing object-based STMs A very similar software technique using best-effort HTM can be used to optimize object-based STM implementations like the one introduced by Herlihy et al. [7] and variants on it [3, 10]. Those implementations use a technique similar to the one described above to effect changes atomically to one or more objects: They make a private copy of the current version of each object to be modified, apply changes to these copies, and finally cause the copies to become current. Again, we can use best-effort HTM to apply the changes *in place* provided we augment the transaction to check that there is not a concurrent operation using the normal method

to effect an update to any of the objects. In the context of Herlihy et al.'s DSTM [7], for example, this is achieved by checking that the object is not currently owned by an active transaction.

This example points out another scenario in which guarantees for very short and simple HTM transactions can be very useful. In particular, the "start block" used in Herlihy et al.'s DSTM introduces a level of indirection that is necessary only because they need to atomically replace a "locator" record, which is larger than can be modified using existing hardware synchronization support. Marathe et al. [10] have showed some clever software techniques for avoiding this level of indirection *in some cases*, but a best-effort HTM that provides a simple guarantee that a short transaction that accesses a single cache line will (eventually) complete obviates the need for this cleverness, and eliminates the indirection in *all* cases. Such guarantees should be quite easy to provide.

5 Summary

We have shown that best-effort HTM can support a variety of useful purposes, including transactions whose size and duration are not limited by the limitations of the best-effort HTM. We therefore believe that best-effort HTM is sufficient to begin the programming revolution that TM has been promising, and that the freedom designers have in providing best-effort HTM (as opposed to the emerging proposals for unbounded HTM) makes it significantly easier for them to commit to supporting TM in hardware.

Whether unbounded HTM designs will ever 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.

Acknowledgements: Numerous people, including at least David Detlefs, David Dice, Steve Heller, Maurice Herlihy, Victor Luchangco, Dan Nussbaum, Nir Shavit, Bob Sproull and Guy Steele, participated in valuable discussions that helped lead to the development of the ideas presented in this paper. Also,Yossi Lev and Peter Damron have been instrumental in the development of our prototype, and this work has helped to clarify many issues related to this work, and to evaluate the many alternatives available in HyTM designs.

References

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

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.

[3] K. Fraser. *Practical Lock-Freedom.* PhD thesis, Cambridge University Technical Report UCAM-CL-TR-579, Cambridge, England, February 2004.

[4] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional

memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[5] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. Oct 2003.

[6] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[7] Maruice Herlihy, Victor Luchangco, Mark Moir, and William 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.

[8] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

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

[10] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. Technical Report TR 868, Computer Science Department, University of Rochester, May 2005.

[11] Jose F. Martinez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 18–29. Oct 2002.

[12] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[13] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228, New York, NY, USA, 1997. ACM Press.

[14] Mark Moir. Transparent support for wait-free transactions. In *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, London, UK, 1997. Springer-Verlag.

[15] Kevin E. Moore, Mark D. Hill, and David A. Wood. Thread-level transactional memory. *Technical Report: CS-TR-2005-1524, Dept. of Computer Sciences, University of Wisconsin*, pages 1–11, Mar 2005.

[16] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305. Dec 2001.

[17] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[18] 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.

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