Split Hardware Transactions

True nesting of transactions using best-effort hardware transactional memory

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

Jan-Willem Maessen
Sun Microsystems Laboratories
JanWillem.Maessen@sun.com

Abstract

Transactional Memory (TM) is on its way to becoming the programming API of choice for writing correct, concurrent, and scalable programs. Hardware TM (HTM) implementations are expected to be significantly faster than pure software TM (STM); however, full hardware support for true closed and open nested transactions is unlikely to be practical.

This paper presents a novel mechanism, the split hardware transaction (SpHT), that uses minimal software support to combine multiple *segments* of an atomic block, each executed using a separate hardware transaction, into one atomic operation. The idea of segmenting transactions can be used for many purposes, including nesting, local retry, orElse, and user-level thread scheduling; in this paper we focus on how it allows linear closed and open nesting of transactions. SpHT overcomes the limited expressive power of best-effort HTM while imposing overheads dramatically lower than STM and preserving useful guarantees such as strong atomicity provided by the underlying HTM.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming

General Terms Algorithms, Performance, Languages

Keywords Transactional Memory, Nesting, Atomicity

1. Introduction

As multicore processors become ubiquitous, application programmers must expose meaningful parallelism in their applications. It is difficult to extract parallelism from traditional lock-based programs; code with fine-grained locking tends to be complicated and prone to subtle latent bugs, but code with coarse-grained locking throws away important parallelism. Transactional Memory (TM) [12, 28] promises to simplify the difficult task of synchronizing concurrent computations. Rather than using locks and condition variables, *transactions* are used to run certain blocks of code designated by the programmer as *atomic blocks*. A transaction appears to execute atomically with respect to other transactions; their actions do not interleave. The critical advantage of TM is that it focuses the programmer on designating *what* computations must run atomically, and leaves the complexity of *how* that atomicity is achieved to the TM implementation itself.

A key feature of TM implementations is the ability to run multiple transactions concurrently, so long as their actions do not *conflict*. Two transactions conflict if one writes to a location accessed (read or written) by the other; we say they are *contending* for this location. In general, it is not possible to determine in advance which locations will be accessed by a transaction when it is run. Conflicts between partially-complete transactions may require a conflicting transaction to *abort*: its effects never become visible to other transactions. The atomic block may be *retried*. A transaction that completes successfully *commits*, making its changes atomically visible to other transactions.

1.1 From Hardware TM to Split Hardware Transactions

TM can be implemented in hardware (HTM) [12, 26, 10, 2, 22], in software (STM) [28, 3, 9, 13, 19], or as in the present paper using a combination of the two [6, 21, 15, 17]. HTM is expected to be faster than the alternatives. Furthermore, most proposed HTM mechanisms offer *strong atomicity*: memory operations within a transaction are guaranteed to be atomic with respect to *all* memory accesses, not merely accesses performed by other transactions. This is a much simpler and more tractable programming model than one that offers only *weak* atomicity, in which the intermediate state of running transactions can be observed and disrupted by non-transactional memory accesses in other threads. Guaranteeing strong atomicity in software effectively requires treating each non-transactional memory access as a miniature single-operation transaction, which is generally considered unacceptable given the overheads of STM. This special treatment must also include memory accesses in preexisting code, which must be either be recompiled using a transaction-aware compiler or must be passed through a binary translator.

However, HTM implementations generally impose strict limitations on program execution. These take two general forms. First, there may be arbitrary *resource limitations* imposed on transactions—for example, the number of locations accessed by a transaction may be limited by buffer space, cache associativity, or other factors. A number of papers suggest hardware mechanisms that overcome resource limitations [2, 26, 22]. Second, there may be *expressiveness limitations*—regardless of resource availability, most HTM mechanisms cannot support *e.g.* true open and closed nesting with nested retry. There are proposals that support true closed nesting [23] and provide assistance to software for concurrency control with *retry* [4], but the trend is to focus on mechanisms that shift policy and scheduling decisions to software.

In this paper we present a novel technique, *Split Hardware Transactions* (SpHT), that overcomes the expressiveness limitations of HTM, achieves substantially better performance than STM, and preserves strong atomicity if it is provided by the underlying HTM.

Copyright is held by Sun Microsystems, Inc.
PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.
ACM 978-1-59593-960-9/08/0002.

SpHT runs each program-level transaction using a *split transaction* by dividing it into multiple *segments*, each of which runs as a separate hardware transaction, and maintaining simple thread-local data to combine segments into a single logical atomic operation. SpHT uses the underlying HTM for conflict detection and atomic commit—the two most complex and expensive aspects of STM algorithms—and this is the key to its performance. This also means that existing transactions which do not take advantage of the new features provided by SpHT can continue to run using HTM exactly as they did before, even if SpHT transactions are running at the same time.

There are other techniques that implement TM using a mix of software and hardware support. Most notable is the Hybrid Transactional Memory (HyTM) of Damron *et al.* [6], in which software transactions are used to execute atomic blocks when attempts to run them using best-effort HTM fail. Unlike SpHT, HyTM is able to overcome resource limitations imposed by the underlying HTM. This comes at a cost: blocks that violate *any* of the restrictions of the underlying HTM, even expressiveness limitations, are run entirely in software. Consequently, every transactional access, even those using HTM, must examine and potentially update shared metadata. Our data suggest that STM techniques are much slower than SpHT, and thus our techniques are preferable when resource bounds are not a problem. In the face of resource limitations, techniques such as Phased TM [17] make it possible to use SpHT to address expressiveness limitations, but to transition smoothly from a mixture of SpHT and HTM to STM or HyTM and back again as resource demands dictate.

1.2 Nesting of Transactions

Nested transactions are a natural consequence of good software engineering: A programmer is given libraries for two shared data structures, and integrates them to yield new atomic actions. We call this *closed nesting*. The simplest implementation of closed nesting is to *flatten* each child transaction into its parent; this can be implemented in any TM system using a simple local counter. However, if any child conflicts with another transaction, the parent must be aborted and rolled back too. We wish instead to provide *true* closed nesting of transactions: when a nested transaction aborts, only the nested computation is retried. True closed nesting scales better than flattening when there is heavy contention toward the end of a long-running transaction: we can put the heavily-contended part in a child transaction which can be retried independently.

More important, true nesting is an important prerequisite for implementing desirable exception semantics. Many proposals for transactional programming language constructs (among them [11, 1]) specify that an exception thrown from inside a running transaction and caught outside the transaction will roll back the effects of the transaction before the exception is delivered. Throwing an exception inside a nested transaction, and catching it in the parent transaction, requires the use of true nesting. Thus, the same issues of composability and ease of engineering that argue for nested transactions also argue for support for true nesting.

Furthermore, true nesting is necessary to support *alternative atomicity*: allowing the programmer to take an alternative action if a transaction aborts (either due to conflict, or through the use of an explicit *abort* construct [11, 4, 1]). For example, the *orElse* construct [11] permits the programmer to specify a *series* of atomic blocks; each is attempted in turn until one succeeds. True nested execution of these alternatives is necessary, otherwise failure of the first alternative will cause failure of the parent and subsequent alternatives will never be attempted. Similar techniques are required for nested use of the *tryAtomic* construct [1], which throws an exception if the enclosed block does not commit.

The chief problem with transactional execution is that state which is updated early in a long-running transaction (often by a now-committed nested transaction) may be required by other running transactions. The TM must choose to either abort the long-running transaction or starve the conflicting actions. An alternative is to provide *open* nesting, in which a nested transaction is performed and that transaction's effects are immediately committed globally, permitting other operations to access the same data structure while the parent transaction is running [24]. A simple example is memory allocation: a call to the allocator can obtain storage from a global pool and commit changes to the global pool immediately, permitting other transactions to allocate storage. The child transaction must register an undo action that is run if the parent transaction aborts; in this case the undo action deallocates the memory. This eliminates spurious conflict when two logically independent actions happen to update common memory locations.

The remainder of this paper is structured as follows: In Section 2 we first give a high-level overview of the SpHT algorithm, and then describe the algorithm and its application to closed nesting in detail. Section 3 extends our basic algorithms to provide user-level abort, alternative actions, and open nesting. Section 4 compares our prototype implementation of SpHT to HTM and STM algorithms. In Section 5 we explore some additional applications of the SpHT mechanism, then we conclude in Section 6.

2. The SpHT Algorithm

The high level idea behind the SpHT algorithm is simple: An atomic block is executed using a *split transaction*, which is a transaction divided into multiple segments. Each segment of the split transaction is executed using a separate hardware transaction. Shared memory writes in a segment are deferred, and are instead logged in a thread-local *write set*. This allows us to commit the hardware transaction executing a segment at any point, without exposing a partial execution of the split transaction to other threads—hence providing *isolation*. Shared memory reads in a segment are logged as well, in a thread local *read set*. A hardware transaction executing one segment can *validate* the reads performed by previous segments by re-reading all the locations in the read set and verifying that the values in these locations are the same as those in the read set. This guarantees *consistency*. Finally, the last hardware transaction copies the written values from the local write set to shared memory; the linearization point [14] of the split transaction is the linearization point of this final hardware transaction. In this way the underlying HTM guarantees *atomicity*. Using HTM to commit atomically is the key to the performance of SpHT. Note that because all writes are deferred until this final segment, SpHT effectively does lazy write acquisition irrespective of the write acquisition technique used by the underlying HTM.

In effect, the read and write set maintained by a split transaction constitute a thread-local log of transaction state that is passed from one segment to the next and updated when transactional memory accesses occur. Both read and write sets must support efficient insertion of new entries and iteration through these entries (for validation and commit respectively); each can be represented as a simple array. Because writes are deferred, a read operation cannot simply read and return the value from shared memory. Instead, it must first look up the address in the write set, and return the most recent value found; the value in shared memory is returned only if the write set lacks an entry for the address. Thus, the write set must also provide efficient address lookup. Because the write set is thread local, this is easily achieved using a hash table or any number of other data structures.

2.1 Segmentation for true nesting

The SpHT algorithm provides the following operations:

```
atomic {                                while (true) {
                                           SpHT_begin()
                                           try {
   a++                                        SpHT_write(&a,1+SpHT_read(&a))
                                              SpHT_pause()
   atomic {                                   while (true) {
                                                 SpHT_begin()
                                                 try {
      foo()                   ⇒                     foo()
                                                 } catch SpHT_invalid { continue }
                                                 SpHT_commit()
                                                 break
   }                                          }
                                              SpHT_resume()
   b++                                        SpHT_write(&b,1+SpHT_read(&b))
                                           } catch SpHT_invalid { continue }
                                           SpHT_commit()
                                           break
}                                       }
```

Figure 1. Translating closed nesting to SpHT. Conflict-free execution is shown in bold.

```
while (true) {
   SpHT_begin()
   try {
      SpHT_write(&a,1+SpHT_read(&a))
      SpHT_pause()
      while (true) {
         SpHT_begin()
         try {
            foo()
         } catch SpHT_invalid { continue }
         SpHT_commit()
         break
      }
      SpHT_resume()
      SpHT_write(&b,1+SpHT_read(&b))
   } catch SpHT_invalid { continue }
   SpHT_commit()
   break
}
```


Figure 2. HTM failure causes re-execution of the current segment

```
while (true) {
   SpHT_begin()
   try {
      SpHT_write(&a,1+SpHT_read(&a))
      SpHT_pause()
      while (true) {
         SpHT_begin()
         try {
            foo()
         } catch SpHT_invalid { continue }
         SpHT_commit()
         break
      }
      SpHT_resume()
      SpHT_write(&b,1+SpHT_read(&b))
   } catch SpHT_invalid { continue }
   SpHT_commit()
   break
}
```


Figure 3. When validation fails, which must happen during an `SpHT_begin` or `SpHT_resume` operation (possibly inside a nested function call such as the call to `foo`), the `SpHT_invalid` exception is thrown and control passes back to the beginning of the enclosing transaction, which is then retried

`SpHT_begin`: Begin a (possibly nested) transaction, starting its first segment.

`SpHT_commit`: Commit a (possibly nested) transaction, ending its last segment.

`SpHT_pause`: Pause a transaction, ending the current segment.

`SpHT_resume`: Resume a transaction, beginning a new segment.

`SpHT_read` and `SpHT_write`: The SpHT transactional read and write operations.

Figure 1 demonstrates simple closed nesting of transactions using SpHT; the user code on the left is compiled to yield the code on the right. A conflict-free execution of this split transaction performs the actions shown in bold. The first call to `SpHT_begin` begins the split transaction and starts the first segment of the outer transaction. This executes `a++` using the `SpHT_read` and `SpHT_write` operations. The segment ends with the call to `SpHT_pause`, and a second call to `SpHT_begin` starts a nested transaction that calls the function `foo`. Since `foo` is part of this split transaction, it must also access memory using the `SpHT_read` and `SpHT_write` operations. When `foo` returns, we commit the child transaction by calling `SpHT_commit`, which ends the last segment of that transaction (`foo` may pause and resume the transaction, so this is not necessarily the segment begun by `SpHT_begin`). Finally, we resume the parent transaction by calling `SpHT_resume`, which starts the last segment. This executes the `b++` statement, and then commits the whole split transaction by calling `SpHT_commit`.

Execution of a segment can fail in two ways. First, the underlying hardware transaction can abort. To simplify the presentation, we assume in this case that the segment is immediately retried: control returns to the operation that began this segment (either `SpHT_begin` or `SpHT_resume`), and a new hardware transaction is started (Figure 2). Second, the validation of the read set performed by `SpHT_begin` and `SpHT_resume` at the start of a segment can fail. In this case the immediately enclosing transaction must abort: no segment is started, and instead the read and write sets are restored to the state they held at the beginning of the enclosing transaction and the `SpHT_invalid` exception is thrown. This causes control to return to the beginning of that transaction, as shown in Figure 3.

If the read set of a parent transaction becomes invalid while a nested transaction is executing, the conflict will immediately cause

```
SpHT_write(addr, value) {
   add ⟨addr, value⟩ to write_set
}
SpHT_read(addr) {
   if (⟨addr, value⟩ in write_set)
      return value
   result = *addr
   add ⟨addr, result⟩ to read_set
   return result
}

SpHT_pause() {
   HTM_commit()
   SpHT_push()
}
SpHT_resume() {
   SpHT_pop()
   while (!HTM_begin()) { ; }
   if (!SpHT_validate())
      HTM_commit()
      SpHT_restore()
      throw SpHT_invalid
}
```

```
SpHT_validate() {
   for ⟨addr, value⟩ in read_set
      if (*addr != value) return false
   return true
}

SpHT_begin() {
   while (!HTM_begin()) { ; }
   if (!SpHT_validate())
      HTM_commit()
      SpHT_pop()
      SpHT_restore()
      throw SpHT_invalid
}
SpHT_commit() {
   if (outermost())
      for ⟨addr, value⟩ in write_set
         *addr = value
      reset read_set and write_set
   HTM_commit()
}
```

Figure 4. Code for the SpHT algorithm

the hardware transaction to fail. On attempting to re-execute the segment, validation will fail and control will pass to the parent transaction. If the conflict was with one of *its* parents' reads, then validation will fail again and control will pass to the next enclosing transaction, and so forth.

Consider the example in Figure 1. Suppose another transaction writes to `a` while running any segment of the transaction in the figure; this will cause the hardware transaction for that segment to fail. If `a` is written during the first segment, it will simply retry from the beginning (the topmost arrow in Figure 2). If the write occurs during the second segment (second arrow in Figure 2), the conflict will be detected when re-executing the nested `SpHT_begin`, causing the `SpHT_invalid` exception to be thrown. The exception is caught by the outermost `try` causing the outer transaction to retry (outer right arrow in Figure 3). Similarly, writing `a` during the final segment will cause the re-execution of `SpHT_resume` to throw the `SpHT_invalid` exception, which will again be caught by the outermost `try` (bottom black arrow in Figure 3).

If a conflict with any memory access done by `foo` occurs during the execution of the final segment, execution will fail in exactly the same way and the outer transaction will be aborted and retried. But if such a conflict occurs while the nested transaction is still running, control will eventually pass back to the nested `SpHT_begin` (middle black arrow in Figure 3); this can occur either through failure of the first segment of the nested transaction, or failure of segments nested within `foo` itself. Upon re-execution no conflict will be found with any previous segment; the inner transaction alone will be retried.

Note that in all of our examples, we use exceptions simply as a convenient way to achieve non-local control flow. Consider, for example, that `foo` in Figure 1 might encounter conflict in the middle of a call to a deeply-nested recursive function. In this case control must be unwound until it reaches the start of the immediately nested transaction; exceptions are a convenient mechanism to achieve this. We are exploring a number of alternative mechanisms, among them maintaining a per-thread "valid" flag to track whether transactional execution is proceeding normally or is in the process of "unwinding" due to conflict. Naturally, different control flow techniques will require different translations from the one we show in this paper.

2.2 Underlying HTM Support

SpHT only assumes basic hardware support that allows beginning and committing a transaction:

`HTM_begin` begins a hardware transaction, returning *true*. If the transaction later aborts, control returns to the `HTM_begin` invocation, which then returns *false*.

`HTM_commit` commits a hardware transaction, making its effects visible globally.

We do not assume any support for non-transactional read and write operations while a transaction takes place, or any support whatsoever for nesting: calling `HTM_begin` while a hardware transaction is running is an error. Similarly, we do not assume that the underlying HTM supports user-level abort of a transaction. In Section 3 we describe how to use SpHT to provide user-level abort even when the underlying HTM does not support it.

2.3 The SpHT Operations

Figure 4 presents pseudocode for the SpHT operations described. `SpHT_write` simply logs the write operation in the write set. `SpHT_read` looks up the read address in the write set, returns the value written if one exists, and otherwise reads the value from shared memory and adds it to the read set.

To ensure that the read and write sets are rolled back to the appropriate state if control flows from a nested transaction to its parent due to a conflict, we maintain a stack of read and write set snapshots. The stack is manipulated by the `SpHT_push`, `SpHT_pop` and `SpHT_restore` operations; `SpHT_push` pushes the current state of the sets on the stack, `SpHT_pop` pops (and discards) the state at the top of the stack, and `SpHT_restore` rolls back the sets to the state at the top of the stack. If we represent the read and write sets as array-based logs then push and pop simply save and restore the index of the next free log entry, a very inexpensive operation. If a hash table is used to support fast write set lookup, the `SpHT_restore` operation must also remove the popped entries from the hash table; this has O(1) amortized complexity. An algorithm for O(1) worst-case write set truncation exists, but increases the constant factor cost of more common lookup and insertion operations.

The SpHT operations maintain the following **Stack State Invariant**:

When the hardware transaction of a segment begins, the state at the top of the stack corresponds to the read and write sets at the beginning of the innermost transaction containing that segment.

We enforce the invariant by unconditionally pushing the current read and write sets using `SpHT_push` whenever we pause a transaction using `SpHT_pause`, and by popping and discarding the saved read and write sets using `SpHT_pop` when we resume a transaction using `SpHT_resume`. These two operations bracket any nested transaction (Figure 1).

Both `SpHT_begin` and `SpHT_resume` start a hardware transaction for the segment they begin, validate the read set by calling `SpHT_validate`, and throw an exception if validation fails (Figure 4). Note, however, the difference in control flow when a conflict is encountered (Figure 3): when encountered by the first segment of a transaction, the *parent* transaction needs to be retried, but for any other segment of a transaction the *current* transaction must be retried. This is why the `try` block of a transaction does not contain the `SpHT_begin` call for its first segment, but does contain the `SpHT_resume` calls for its remaining segments. Thus, as shown in Figure 4, when `SpHT_resume` fails validation, the read and write sets are rolled back to the state at the beginning of the current transaction, which by the Stack State Invariant is at the top of the stack. But when `SpHT_begin` fails validation, we first discard the state at the top of the stack (that corresponds to the current transaction), and only then roll back the read and write sets to the older state that is now at the top of the stack.

To commit a split transaction, `SpHT_commit` commits the currently executing hardware transaction. For the outermost transaction, however, the write set must first be written back to memory. If the write set contains multiple entries for a particular address, the final value must be the most recent one written. There are various ways to determine whether a transaction is outermost; one is to note that the state stack is empty.

Finally, note that we never manipulate the snapshot stack or throw an exception while a hardware transaction is running; the translation in Figure 1 always runs non-transactionally whenever control passes into or out of a `while` loop or an exception is thrown. By contrast, a hardware transaction is running whenever normal control flow enters or exits a `try` block.

2.4 Pragmatics

Since all modifications to the read and write sets while a segment is executing are done as part of a hardware transaction, if that transaction is aborted, these modifications are not committed. Thus, in order to retry the most recent segment it is not necessary to restore the read and write sets from the stack. Our translation reflects this fact: it first retries the most recent segment when the hardware transaction fails; if this segment is the first segment of the transaction, it avoids the `SpHT_restore` operation as it does so.

Note also that in certain cases it is acceptable to retry the most recent segment when the hardware transaction fails, but to abort the entire split transaction when encountering a conflict with a prior segment (that is, when an `SpHT_validate` operation fails). This allows us to retry any single-segment leaf transaction (that is, one that is not a parent of another transaction), but otherwise requires aborting the outermost enclosing atomic block when a conflict with a parent occurs. To support this, there is no need to maintain the state stack for the read and write sets — the most recent segment can be retried, and if a conflict with a previous segment is detected, the whole split transaction is retried with fresh (empty) read and write sets. In particular, if there is only one level of transactional nesting, and *every* nested transaction is a single segment, the read and write sets need never be restored explicitly—we obtain true nesting without the need for the stack mechanism.

For similar reasons, the last segment of the outermost transaction can perform its reads and writes directly rather than logging them: they will be rolled back anyway if the hardware transaction fails. Combining this with the previous optimization, a single-segment non-nested transaction can be executed using HTM directly without any of the mechanisms of SpHT, even when another thread is executing a split transaction.

As described, if a nested transaction fails to validate its read set, layers of nesting are peeled off one at a time; at each level a new hardware transaction is started and the read set is re-traversed. This incurs $O(rd)$ overhead in the worst case, where r is the number of reads and d is the depth of nesting. We can maintain a "water mark" indicating how far into the read set a conflict was found and use this to quickly abort multiple nested transactions, reducing the cost of re-validation to $O(r)$. In practice, however, d is generally small; the extra record keeping complicates control flow and is not likely to pay off.

For a transaction with a large read set and many small segments the cost of re-validating the read set at the start of each segment may dominate the cost of actually executing a segment. This is a problem with any TM algorithm which must use software to perform eager validation. Ennals [8] advocates avoiding validation and detecting inconsistencies mainly using timeouts and error handlers. While in general we do not believe in this approach, a compiler may be able to determine that a particular ***segment*** of a transaction can execute without requiring consistency; in this case, the read validation for that segment can be skipped.

In this paper we do not treat transactional contention management [27] in any detail; in practice, of course, contention between transactions must be handled gracefully or livelock may occur. Recall that there are two kinds of failure when a split transaction is running: the underlying hardware transaction can fail, or validation of the read set can fail. In the former case, the simple HTM we have assumed returns control to software (the `while` loops in `SpHT_begin` and `SpHT_resume`, Figure 4). Note, however, that at this point the next action will be to start a new hardware transaction and re-validate the read set; in the presence of contention we expect validation to fail immediately thereafter. Thus, relatively minimal contention management is necessary at this point, directed primarily at the case in which contention on an already-validated location in the read set causes repeated HTM failures during validation. More important is the contention management that must occur when validation fails (before the calls to `SpHT_restore` in `SpHT_begin` and `SpHT_resume`, Figure 4). We can choose any contention management mechanism suitable for a system with invisible reads, eager conflict detection, and lazy write acquisition; note in particular that contention management decisions can be made based upon the contents of the read and write sets. This is a great deal more information than a simple best-effort HTM can provide, and as a result we expect SpHT to permit more sophisticated contention management policies than the underlying HTM would.

Finally, note that large transactions can overflow the read or write set, requiring it to be enlarged. This is best done outside a hardware transaction, so that the larger set can be re-used if the transaction fails. This is a simple matter: when overflow is detected (in `SpHT_read` or `SpHT_write`) we call `SpHT_pause`, resize the set, and call `SpHT_resume`.

3. Beyond Closed Nesting

User-level abort in the style of the *retry* construct [11, 4, 1] can be added to SpHT without changing the implementation. A call to `SpHT_abort` abandons and retries execution of the current transaction. It looks just like the failure path in `SpHT_resume`, calling `HTM_commit` and `SpHT_restore`, and finally throwing `SpHT_invalid`. In practice, of course, contention managers often

```
atomic {
    transactional block 1
} else {
    transactional block 2
}

        ⇓

SpHT_pause() /* if nested */
while (true) {
   if (SpHT_begin()) {
      try {
         translated block 1
      } catch SpHT_invalid { goto block2 }
      SpHT_commit()
      break
   }
 block2:
   if (SpHT_begin()) {
      try {
         translated block 2
      } catch SpHT_invalid { continue }
      SpHT_commit()
      break
   }
}
SpHT_resume() /* if nested */
```

Figure 5. Translation of alternative blocks

need to distinguish failure due to SpHT_abort from validation failure due to conflict. This can be as simple as throwing a different type of exception for a call to SpHT_abort. The read and write set of the aborted split transaction capture the information necessary to provide condition-variable style blocking after an abort [11, 4].

We can extend our model to encompass transactions with *alternative blocks* separated by else, similar in spirit to *orElse* [11]: the alternative blocks are attempted one at a time, proceeding to the next one if the current one aborts. If all the alternatives abort the entire atomic construct is retried. The translation shown in Figure 5 uses a variant of the operations shown in Figure 4; we simply change SpHT_begin to return false when the hardware transaction fails, instead of implicitly starting a new one. We move on to the next alternative if one is available, or simply call SpHT_begin again for an ordinary atomic block.

We can also use SpHT to implement open nesting of transactions [24]. An open nested transaction commits its effects to memory immediately, rather than deferring them until its parent commits; the memory locations accessed by the open nested transaction may immediately be accessed by other transactions. If the parent of an open nested transaction later aborts, the effects of the open nested transaction may need to be undone. This is done by permitting an open nested transaction to register an *undo* action which is run if the parent aborts (this undo action may itself run as an open nested transaction). The chief complication in implementing open nesting is defining what happens when an open nested transaction accesses state that was previously accessed by its parent. Briefly, the open nested transaction can appear to operate on a state that is a mixture of the state at its linearization point and a state that appears to occur *later* in time, when the parent transaction commits [24]. If we forbid such overlap, running an open nested transaction under SpHT is simple: Pause the parent transaction, then run the open nested transaction using SpHT with fresh read and write sets or using HTM directly, and finally resume the parent transaction. When the open nested transaction commits, its effects are immediately made visible globally. Forbidding overlap between an open nested transaction and its parent is the most restrictive choice of semantics, but seems to be the most desirable one in practice; however, any other semantics can be provided simply by permitting the open nested transaction to access the read and write sets of the parent transaction.

4. The SpHT Prototype and Evaluation

As described, the SpHT mechanism uses best effort HTM to provide true nesting of transactions, with the additional overhead of maintaining a read and a write log in software. In this section, we evaluate the most significant sources of overhead by comparing executions of two microbenchmarks using a variant of the LogTM HTM [22], using an SpHT prototype that implements most of the SpHT functionality with this HTM, and using TL2 [7]—a state of the art STM.[1]

4.1 The SpHT Prototype

Our SpHT prototype, implemented in C++, provides the basic functionality described in Section 2. Each thread is allocated an SpHTHandler object which maintains the read and the write sets for that thread. Each set consists of entries stored in chronological order in an array. An open-addressing hash table is used for address lookup; each write set entry contains a link to earlier entries in the same hash bucket. A 64-bit Bloom filter speeds up write set membership tests. This allows for quick address lookup by SpHT_read, as well as fast copy-back of values to shared memory when a split transaction commits.

Our prototype implementation does not support the full functionality of SpHT: we have not implemented the nesting stack. As described in Section 2.4, the stack mechanism is not necessary as long as every nested transaction consists of a single segment (this limits the nesting depth to at most one). We expect the cost of maintaining the stack to be small compared to the overhead of maintaining the read and write sets, as these are accessed and frequently modified by every read and write operation of the atomic block, whereas the stack is only manipulated when entering and leaving a nested atomic block. Moreover we expect stack manipulation to be a relatively inexpensive operation: push and pop simply move two log indices on and off the stack; restore occurs only when a nested transaction aborts, and needs only to traverse the writes performed by that nested transaction. We therefore believe that our prototype reflects the bulk of the SpHT's overhead over HTM, and that it suffices for demonstrating the use of SpHT for true nesting of transactions.

4.2 Simulated HTM Support

To simulate the underlying HTM, we use a variant of the Wisconsin LogTM simulator [22], a multiprocessor simulator based on Virtutech Simics [18] extended with customized memory models by Wisconsin GEMS [20], and further extended to simulate the unbounded LogTM architecture [22]. By default the LogTM architecture handles contention completely in hardware, transparently retrying failed transactions; by contrast, the SpHT algorithm expects to be notified upon transaction failure. In order to simulate a simpler best-effort HTM, we collected all our data using the variant of the LogTM simulator described in [6, 17]. Rather than handling failure in hardware, this version branches to a *failure address* provided when the transaction began; we use this mechanism to implement the HTM_begin method described in Section 2. The systems we simulate share the same multiprocessor architecture described in [22], except that our simulated processors were run at 1.2GHz, not 1GHz, and we use the simpler MESI_SMP_LogTM cache coherence protocol, instead of the MOESI_SMP_LogTM protocol.

[1] We used TL2 as a pure software solution to compare against rather than the STM component of HyTM [6], as the latter is designed to run concurrently with hardware transactions, incurring overhead that pure STM solutions can avoid.

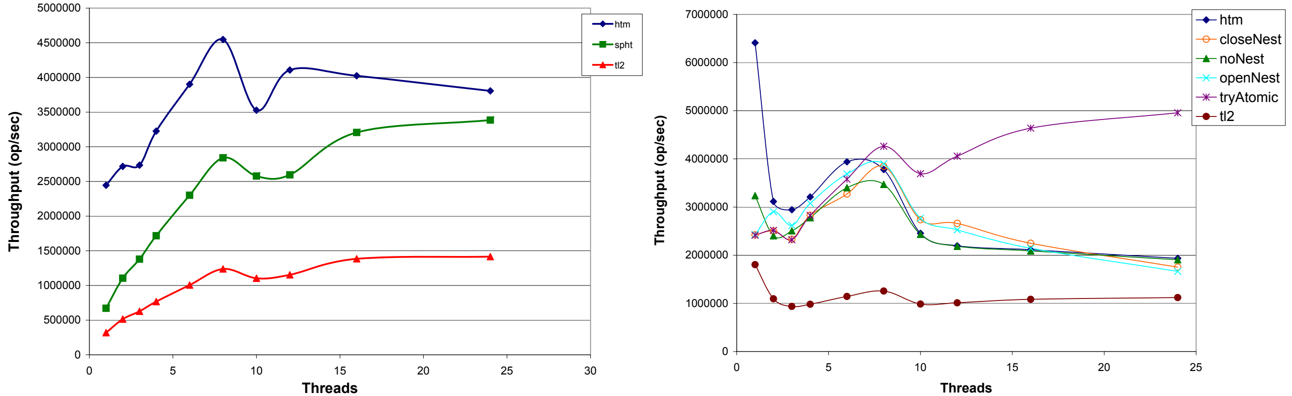

Figure 6. Performance results for non-nested Red-Black tree (left) and nested bank account (right).

We use this simulator to evaluate three TM mechanisms: the above version of LogTM, SpHT using this HTM, and TL2 [7]. For a fair comparison with hardware transactions, we allow the compiler to inline the transactional read and write methods of both SpHT and TL2. In both the pure HTM and SpHT experiments, we used a simple exponential backoff scheme to back off before retrying a failed hardware transaction (note that with split transactions, the backoff is done separately per segment).

4.3 Experiments Without Nesting

Our first group of experiments simply evaluates the relative overheads of HTM, SpHT, and TL2 in the absence of nesting. Our micro-benchmark initializes a red-black tree with 2000 unique keys chosen at random from the range 0 to 4095. We then measure the total throughput as each thread performs a random mix of 10% insertion, 10% deletion, and 80% probe operations on random keys. Each operation is run as a separate atomic transaction. We choose a relatively small key space and a moderate rate of insertion and deletion in order to increase contention between transactions. The single threaded run demonstrates the relative contention-free overhead of the three mechanisms. Runs at larger thread counts demonstrate the benefit of using hardware for conflict detection and atomicity, which is one of the main advantages of SpHT over pure STM solutions.

Figure 6 shows the results of the experiment. As can be seen, the SpHT mechanism achieves substantially higher throughput than that achieved by TL2. That gap increases with the level of contention, starting as a factor of 2.1 with a single thread, and growing to a factor of 2.4 with 24 threads. The gap between HTM and and SpHT, on the other hand, decreases with the level of contention, starting at a factor of 3.6 with a single thread, and dropping to a factor of 1.12 with 24 threads.

These results are not surprising: the SpHT mechanism uses the underlying HTM to handle conflicts, and hence only induces *local* overhead (to handle the read and write sets) which is independent of the contention level. Thus with single threaded execution SpHT only closes a small fraction of the gap between STM and HTM, but as contention rises performance gets closer to that of HTM.

Finally, note that there is a throughput decrease with all TM mechanisms between 8 and 10 threads. This is because the 10 thread experiment runs on a simulated 16 processor machine, while the 8 thread experiment runs on a simulated 8 processor machine. The more processors the simulated machine has, the slower the interconnection between them, and hence the more we pay for every cache miss incurred by contention. Unfortunately, since the simulation overhead increases with the size of the simulated machine (the simulator runs single threaded), we did not have the resources to simulate all our runs on the largest (32 processor) machine required for the 24 threads run. The same argument explains the reduction in HTM throughput between 2 and 3 threads.

4.4 Comparing Approaches to Nesting

Our second group of experiments explores how split transactions perform in situations where nesting might be valuable. We focus on showing how true nesting enriches the programming model, and how the richer programming model may allow better handling of contention in a composable manner.

Unfortunately, we are not aware of any readily-available benchmarks making use of nested transactions. (This is probably because many TM implementations do not yet support true nesting of transactions.) Moreover, in the absence of compiler support for nesting atomic blocks, we were limited to writing a short and simple micro-benchmark that we could hand-program to resemble the output we would expect from such a compiler.

In our micro-benchmark threads operate on a small number of bank accounts (ten in this case). Account balances are spread among multiple sites: a central site and a few local ones. Our goal is to simulate a situation in which threads work on lightly-contended (but shared) data, but periodically update the highly-contended central site. The balance of an account is the sum of its balances at all sites. Each site is represented by a simple array of account balances.

Threads are distributed evenly between the local sites, and can execute one of two atomic operations on an account: a deposit or a withdrawal. To deposit money, a thread reads the account balance in its local site, and if the new balance after the deposit exceeds some threshold, transfers half of the new balance to the central site. Similarly, to withdraw money, a thread computes the new balance in the local site, and if this balance is negative, it transfers twice that balance from the central to the local site. The central balance is permitted to be negative or arbitrarily large.

In our experiments, the number of local sites is half the number of threads. Therefore, as we increase the number of threads local contention is constant but the contention on the central site increases. We measure throughput as threads perform a random series of operations. Each operation chooses a random account and deposits or withdraws a random amount, with a 50% probability of each operation. We compared six methods:

htm, tl2 and spht-noNest: In these experiments, the withdraw and deposit methods are each implemented using a single atomic block that is executed by an HTM, TL2 or SpHT transaction, respectively.

spht-closeNest: In this experiment, the withdraw and deposit operations transfer money between the central and local sites using a closed nested transaction. The nested transaction is retried until it succeeds (or until a conflict in the parent transaction is detected).

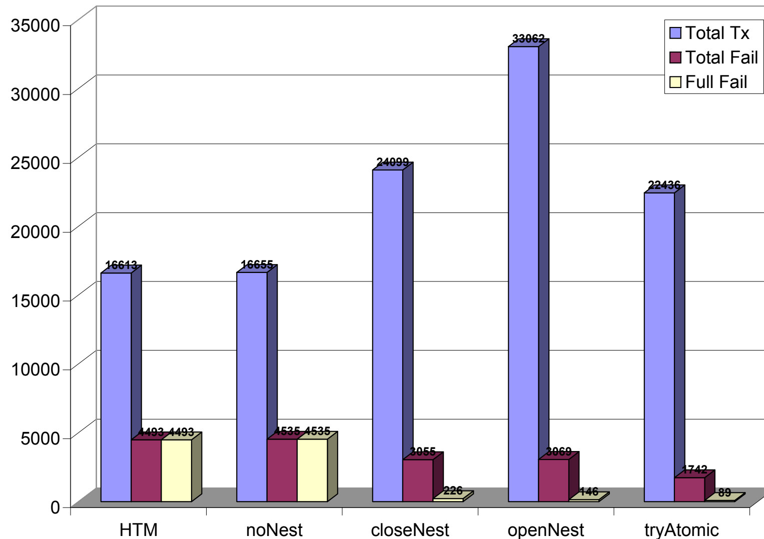

Figure 7. Abort behavior of nested bank account benchmark with 12 threads. Vertical axis is number of transactions.

spht-tryAtomic: As with the spht-closeNest experiment, access to the central site uses a closed nested transaction. In this variation, however, if the nested transaction that transfers money to the central account in a deposit operation aborts, we do not retry it, but instead leave the money in the local site. Nested transactions executed by the withdraw operation are still retried until they succeed—a local account balance is never negative. Thus, when an account in the central site is contended, we keep more money in the local sites, thus reducing contention on the central site.

spht-openNest: In this experiment, the withdraw and deposit operations access the central site using an open nested transaction. We pause the split transaction executed by the withdraw or deposit operation, update the central account using a separate hardware transaction, and then resume the split transaction to update the local site with the new balance. If the split transaction later aborts, we undo the update to the central site using another hardware transaction, run by the catch block that handles the `SpHT_invalid` exception that aborts the split transaction.

Figure 6 presents the throughput achieved by each of these experiments. Note that with 10 threads and above, HTM loses all its benefit over SpHT; the methods that achieve the best performance are the ones that best handle contention. As can be clearly seen, the tryAtomic method achieves the best throughput for any experiment with 8 threads or above. This is not surprising, as it is the only method that exploits the stronger semantics provided by true nesting to *reduce* the contention on the central site when it is detected; the other methods merely *isolate* the rest of the operation from this contention. This isolation does help to boost performance a bit when the level of contention is moderate, but with extremely high levels of contention eventually all methods are contention bound, and the overhead of additional hardware transactions and validation operations erases the benefits this isolation provides over non-nested transactions.

To further analyze the differences between the methods, we take the 12-thread runs and examine how many hardware transactions were executed by each method, how many of them aborted, and how many of these aborts resulted in a full retry of the outer atomic block. Figure 7 presents these results. In the figure, the left bar (blue) in each category shows the total number of hardware transactions executed, the middle bar (red) shows how many of them aborted, and the right bar (ivory) is the number of times the whole atomic block was retried.

When running with HTM or with non-nested SpHT the total number of aborts is roughly the same, and all of them result in a full retry of the atomic block (as neither method supports true nesting). Both closed and open nesting reduce the total number of aborts by about a third (because the transaction that accesses the central account is now shorter), but most of the savings are in the total number of full retries of the atomic block: in both cases, less than 15% of the aborts result in aborting the whole split transaction; the rest retry only the aborted segment. Thus accessing the central site in a nested transaction clearly isolates the operation as a whole from contention. Also note that the total number of transactions executed by the openNest method is significantly higher than that of the closeNest method; this is simply because the closeNest method only runs two segments per split transaction (the second segment is a transfer—that is, is updates both the parent and the local site), while the openNest method runs an open nested hardware transaction on the central site in addition to the two segments that operate on the local balance (and possibly also runs a fourth transaction to clean up the central site update when the whole split transaction fails). Finally, the tryAtomic method is a clear win: it has a noticeably lower total number of aborted hardware transactions, and as a result a proportionally lower number of full retries of the split transaction.

5. Extensions and New Applications

The most fundamental shortcoming of SpHT is its inability to overcome resource limitations imposed by the underlying HTM; indeed, the need to maintain the read and write sets may exacerbate these limitations. However, HTM that is resource-limited only by L1 cache size can run most transactions, and HTM that is resource-limited by L2 cache can run nearly all transactions [5]. Phased Transactional Memory (PhTM) [17] is a promising technique in this case: most of the time the program can use a mix of HTM and SpHT for transactional execution. When a running transaction exceeds these bounds, the whole system can make a "phase change" and run in software or hybrid mode [6], then change back again afterwards. Moreover, SpHT permits different segments of a single transaction to be run in different phases, using different transactional algorithms.

Segmented transactions are a flexible and open-ended mechanism, and we are exploring a number of additional uses for them. Although the operations executed while the SpHT is paused are not part of its atomic operation, they can benefit from the fact that all memory operations done by the split transaction so far are logged in the read and write sets. Using a split transaction, a debugger can insert a breakpoint in an atomic block and permit the programmer to inspect its state. In a recent paper [16], Lev and Moir describe many other debugging techniques for transactional programs that require access to the transaction's read and write set—many of these techniques can be implemented using SpHT, without requiring the debugged atomic block to be executed using STM.

Thread-level speculative execution of loops [25] is closely related to TM, and the two can use many of the same mechanisms [10]; however special support (in the form of non-transactional reads and writes or in the form of ordered commit) is required in order to use HTM to speculatively parallelize loops. In the absence of this support, we can use SpHT instead: pause each iteration just before completion, block until previous iterations commit, then resume, check for conflicts, and commit the split transaction.

The Fortress programming language [1] provides atomic blocks, and permits (but does not require) concurrent execution of multiple threads *within* a transaction. These threads can communicate using nested transactions. We have sketched a number of techniques that use SpHT to provide embedded transactional parallelism, and we are exploring tradeoffs among them. One of the simplest works much like speculative loop execution: threads execute in parallel using the parent read and write sets as a starting point. If two threads conflict, the effects of one are retained and the other is re-run. This provides the appearance that the threads ran sequentially in some order, but allows the order to be chosen at run time to avoid conflict.

6. Conclusion

Transactional memory (TM) is a promising paradigm to allow programmers to write correct and scalable concurrent programs. While hardware transactional memory is expected to run significantly faster than software-only TM, hardware is also expected to have various limitations; in particular most of the practical hardware TM proposals do not support true nesting of transactions—a desirable property for *composability* in concurrent programs that use TM.

In this paper we described the Split Hardware Transaction (SpHT) mechanism, which uses minimal software support to combine multiple hardware transactions into one logically atomic operation. We showed how to use this mechanism to provide true nesting of transactions when nesting is not supported by the underlying hardware TM. We presented preliminary experimental results demonstrating that transactions executed using SpHT are faster, and scale much better, than ones executed using software transactions, and that the stronger semantics provided by true nesting can help in better handling contention and hence boosting performance.

Finally, SpHT can be combined with the HyTM and the PhTM approaches, allowing split transactions to be executed concurrently with hardware and software transactions. Together, the mechanisms use simple best-effort hardware transactional memory to provide the programmer with a better and more flexible transactional programming model than is possible with simple hardware alone, while providing better performance and scalability than pure software transactional memory.

Acknowledgments

The authors would like to thank Maurice Herlihy and the Scalable Synchronization and Programming Languages Research Groups at Sun Microsystems Laboratories for conversations and advice leading up to this work, and the indulgence to see it to completion. Special thanks are due to Dan Nussbaum for his hard work and good advice, without which we would not have simulation results.

References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Sam Tobin-Hochstadt. The fortress language specification, version 1.0β. Available from http://research.sun.com/projects/plrg/Publications/, March 2007.

[2] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, San Franscisco, California, February 2005.

[3] C. Scott Ananian and Martin Rinard. Efficient object-based software transactions. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, October 2005.

[4] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle 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] JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, Christos Kozyrakis, , and Kunle Olukotun. The common case transactional behavior of multithreaded programs. In *12th International Symposium on High Performance Computer Architecture (HPCA)*. Feb 2006.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. 12th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2006.

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

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

[9] Keir Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[10] 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.

[11] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[12] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[13] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, Jul 2003.

[14] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions On Programming Languages and Systems*, 12(3):463–492, July 1990.

[15] Sanjeev Kumar, Michael Chu, Christopher Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Preceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[16] Yossi Lev and Mark Moir. Debugging with transactional memory.

Transact 2006 workshop, June 2006. http://research.sun.com/scalable/pubs/Lev-Moir-Debugging-2006.pdf.

[17] Yossi Lev, Mark Moir, and Dan Nussbaum. Phtm: Phased transactional memory. Transact 2007 workshop, August 2007. http://www.cs.rochester.edu/meetings/TRANSACT07/papers/lev.pdf.

[18] 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.

[19] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. Technical report, Cracow, Poland, Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May 2005.

[20] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[21] Mark Moir. Hybrid transactional memory, Jul 2005. http://www.cs.wisc.edu/trans-memory/misc-papers/moir:hybrid-tm:tr:2005.pdf.

[22] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[23] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, New York, NY, USA, 2006. ACM Press.

[24] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.

[25] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[26] 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.

[27] 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.

[28] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.