Transaction-Friendly Condition Variables[*]

Chao Wang
Lehigh University
chw412@lehigh.edu

Yujie Liu
Lehigh University
lyj@lehigh.edu

Michael Spear
Lehigh University
spear@cse.lehigh.edu

ABSTRACT

Recent microprocessors and compilers have added support for transactional memory (TM). While state-of-the-art TM systems allow the replacement of lock-based critical sections with scalable, optimistic transactions, there is not yet an acceptable mechanism for supporting the use of condition variables in transactions.

We introduce a new implementation of condition variables, which uses transactions internally, which can be used from within both transactions and lock-based critical sections, and which is compatible with existing C/C++ interfaces for condition synchronization. By moving most of the mechanism for condition synchronization into user-space, our condition variables have low overhead and permit flexible interfaces that can avoid some of the pitfalls of traditional condition variables. Performance evaluation on an unmodified PARSEC benchmark suite shows equivalent performance to lock-based code, and our transactional condition variables also make it possible to replace all locks in PARSEC with transactions.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

Keywords

Transactional Memory, Semaphore, Condition Synchronization

1 Introduction

The current Draft C++ TM Specification [1] is particularly well-suited to transactionalizing legacy code: one need only replace lock-based critical sections with lexically scoped transactions, and then annotate functions that are called by the transaction. This effort is sufficient for simple critical sections (even those performing I/O), but fails when there is condition synchronization. Both Ruan et al. [17] and Skyrme and Rodriguez [19] observed this problem in the course of their efforts to transactionalize memcached and luaproc, respectively.

In C++, a condition variable must be associated with a named mutex. A thread must hold the mutex to WAIT on the condition variable, and the act of waiting effectively completes one critical section. The thread can then be put to sleep and another thread may safely acquire the mutex and modify shared state. If its modifications satisfy a programmer-specified predicate, it must NOTIFY the condition variable. The waiting thread can then be woken, at which point it attempts to acquire the mutex to execute the continuation of its critical section.

In existing concurrent programs, the act of breaking atomicity at the point of a thread's WAIT does not compromise correctness: the programmer is responsible for checking and restoring invariants when the thread resumes execution. Thus a straightforward translation of the synchronizing critical section to a pair of transactions is not, itself, a concern. The problem is that a waiting thread must release the lock and put itself to sleep in a single atomic operation; if the sleep is delayed until after the lock is released, then it is possible for an intervening NOTIFY to "miss" the waiting thread.

Traditionally this problem is solved by implementing the locking and waiting mechanisms within the operating system (OS). In this manner, the OS is able to mark the thread as waiting, release the lock, and schedule another thread in a way that appears atomic with respect to all other threads. A common practice in this case is to relax the guarantees made by the OS: in Mesa, a NOTIFY could accidentally wake more than one thread [4], and in the POSIX specification, a WAIT can return without being paired with a NOTIFY. It should be noted that modern code accepts these relaxations, and employs a few simple patterns (i.e., calling WAIT within a `while` loop) to overcome any potential spurious wake-ups.

In this paper, we present a novel implementation of condition variables that is compatible with both locks and TM. Our work is inspired by prior work by Dudnik and Swift [6], which discusses an extension to the Solaris OS that supports condition variables within a research hardware TM prototype, and AtomCaml [16], which was the first to consider splitting WAIT operations within transactions. The key innovation in our work is that each condition variable is implemented as a transactional queue of per-thread counting semaphores. This design avoids several pitfalls identified by Birrell [3], and makes it possible to implement condition variables portably, without OS modification. The result is *simpler* and *more flexible* than traditional condition variables, does not have a noticeable impact on performance, and is agnostic to the TM implementation (i.e., it is compatible with both hardware and software transactions). Furthermore, our implementation is immune to the spurious wake-ups that can occur when condition variables are implemented within the OS.

[*]This work was supported in part by the National Science Foundation under grants CNS-1016828, CCF-1218530, and CAREER-1253362.

Algorithm 1: The CondVar Specification

```
shared states
  $Q$   : Set<Thread>   // waiting threads; initially $\emptyset$
// $p$ refers to the thread that performs the operation
procedure WAITSTEP1()
    $Q \leftarrow Q \cup \{p\}$

function WAITSTEP2() : Boolean
    return $p \in Q$

procedure NOTIFYONE()
    if $\exists\, x \in Q$ then $Q \leftarrow Q \setminus \{x\}$

procedure NOTIFYALL()
    $Q \leftarrow \emptyset$
```

The remainder of this paper is organized as follows. Section 2 presents our condition variable algorithm. Section 3 discusses our implementation in C++, and Section 4 discusses implementation challenges for a transactionalizing compiler and runtime system. In Section 5, we evaluate our algorithm using both lock-based and transactionalized versions of the PARSEC [2] benchmark suite. In Section 6, we review related work, with a focus on programming models. Section 7 discusses future work and concludes.

2 Specification and Algorithm

Unlike linearizable concurrent data structures, a condition variable does not naively admit a sequential specification: an invocation of the WAIT method cannot produce a response without it paring with an intervening operation (a NOTIFY) by another thread. Nonetheless, we require a specification of the behavior of condition variables before we can describe an algorithm that can be proven correct. To that end, we begin by presenting an abstract specification modeled after that proposed by Birrell et al. [4], and the sequential specification of a lower-level CondVar object. Using this specification, we introduce a generic algorithm and prove that it is both correct and immune to spurious wake-ups. We then discuss other desirable properties of the algorithm.

2.1 Conventional Specification

In order for a condition variable to be useful, a programmer must be able to reason about the order in which WAIT and NOTIFY operations are performed. Traditionally, this is accomplished by coupling the use of condition variables with mutual exclusion locks.

Birrell et al. [4] proposed a semantics that is widely adopted by many implementations of condition variables. In Birrell's specification, the abstract states of a condition variable object consist of a set Q of waiting threads (initially empty) and a mutual exclusion lock L. The object supports three operations specified as follows:

- A WAIT operation must be invoked within a critical section where L is held by the invoking thread p. The operation consists of two *separate* atomic steps: the first step adds p to Q and releases L atomically; in the second step, the thread is suspended until reaching a state where $p \notin Q$ and L is not acquired, at which point it acquires L and returns.
- A NOTIFY operation atomically removes some non-empty proper subset of threads from Q if Q is not empty.
- A NOTIFYALL operation atomically makes Q empty.

2.2 A Common Specification

Our aim is to produce a specification of condition variables that is compatible with both locks and transactions. To this end, we need to eliminate the notion of locks from the specification. We also notice that in Birrell's specification, NOTIFY can be simply implemented as a NOTIFYALL, and thus, we replace the former operation with a NOTIFYONE operation to allow removing exactly one thread from the set of waiting threads.

We introduce *atomic sequences* as the foundation of our common specification. An atomic sequence $\langle S \rangle$ is a dynamic sequence of instructions S executed by some thread p, which are enclosed by special beginning and ending instructions. The sequence of instructions S in $\langle S \rangle$ is executed atomically if there is no occurrence of a WAIT operation in S. Atomic sequences are flat-nested, that is, a nested atomic sequence $\langle S_0 \ ; \ \langle S_1 \rangle \ ; \ S_2 \rangle$ is semantically equivalent to $\langle S_0 \ ; \ S_1 \ ; \ S_2 \rangle$.

A WAIT operation can appear only in an atomic sequence. An atomic sequence $\langle S;\text{WAIT};C \rangle$, where S is the preceding sequence before the first occurrence of WAIT and C is the continuation sequence after the WAIT operation, is semantically equivalent to the following sequence:

$$\langle S \ ; \ Q \leftarrow Q \cup \{p\} \rangle \ ; \ \langle \textbf{assert}\ p \notin Q \rangle \ ; \ \langle C \rangle$$

A NOTIFYONE or NOTIFYALL operation can appear either in an atomic sequence or not. In either case, a NOTIFYONE is equivalent to $\langle \textbf{if}\ \exists x \in Q\ \textbf{then}\ Q \leftarrow Q \setminus \{x\} \rangle$, and a NOTIFYALL is equivalent to $\langle Q \leftarrow \emptyset \rangle$.

We extract the interface of a CondVar object from the above specification. The interface consists of four operations listed in Algorithm 1: WAITSTEP1, WAITSTEP2, NOTIFYONE, and NOTIFYALL. Intuitively, a WAIT operation (Step1 and Step2) adds the caller thread to the waiting set and suspends the caller. A NOTIFYONE wakes a thread that has performed WAITSTEP1 on the same CondVar but has not yet been woken, and NOTIFYALL wakes all threads that have performed a WAITSTEP1 on a CondVar but have not yet been woken.

We define the set of *legal* histories by imposing constraints on the set of all sequential histories permitted by the CondVar object in Algorithm 1.

DEFINITION 1. *A sequential history of a CondVar object is legal if it satisfies the following:*

(1) *For every thread, a* WAITSTEP1 *operation is immediately followed by a* WAITSTEP2 *in the thread's history.*
(2) *Every* WAITSTEP2 *operation returns false.*

2.3 A Generic Implementation

Let us now consider an implementation of condition variables that satisfies this specification. We represent each condition variable as a set of thread identifiers, and additionally require per-thread flags. The set stores the identities of all threads waiting on a particular CondVar; the flags are a convenience mechanism that provides a means for decoupling set operations from the instructions that allow waiting threads to continue.

In this algorithm, a thread performs WAITSTEP1 by setting its flag and then inserting its unique identifier into a particular CondVar's set. To wake a thread, should there be one waiting, a thread uses NOTIFYONE to remove one entry from the set, and then clears the corresponding thread's flag. The NOTIFYALL method is similar to NOTIFYONE, except it wakes all threads that are sleeping on the CondVar.

We now prove a few properties of this generic implementation. For the proofs, we assume that each line in the code listing is exe-

Algorithm 2: A Generic CondVar Implementation

```
shared states
  $Q$       : Set<Thread>   // waiting threads; initially $\emptyset$
  $spin_p$  : Boolean       // per-thread flag; initially false

procedure WaitStep1()
1   $spin_p \leftarrow$ true
2   $Q \leftarrow Q \cup \{p\}$

function WaitStep2() : Boolean
3   while true do if $\neg spin_p$ then return false

procedure NotifyOne()
    // remove from $Q$ an arbitrary element $x$ if exists
4   if $\exists\, x \in Q$ then $\{Q \leftarrow Q \setminus \{x\}; e \leftarrow$ true$\}$ else
    $e \leftarrow$ false
    // clear $spin_x$ if some $x$ is removed from $Q$ by last step
5   if $e$ then $spin_x \leftarrow$ false

procedure NotifyAll()
    // move all elements from $Q$ to $Q'$
6   $\langle Q' \leftarrow Q\ ;\ Q \leftarrow \emptyset \rangle$
    // remove some $x$ from $Q'$ and clear $spin_x$
7   while $\exists x \in Q'$ do $\{Q' \leftarrow Q' \setminus \{x\}; spin_x \leftarrow$ false$\}$
```

cuted as an atomic step. Note that for the while-loops at lines 3 and 7, "executing as an atomic step" means executing one iteration of the loop as an atomic step, including the evaluation of the condition and at most one execution of the loop body. We use the notation $p@k$ to denote that thread p is about to execute the step at line k.

The main obligation of the proof is to show that there exists a refinement mapping from the generic implementation to the CondVar specification. The following invariants capture the basic properties of the algorithm, which can be proved together (as one conjunction) by induction over reachable states.

LEMMA 2. *The following statements hold as invariants:*
(1) $p@1 \implies \neg spin_p$
(2) $p@2 \implies spin_p$
(3) $p \in Q \implies p@3 \wedge spin_p$
(4) $p@5 \wedge e \implies x@3 \wedge spin_x$
(5) $p@7 \wedge x \in Q' \implies x@3 \wedge spin_x$

THEOREM 3. *The generic CondVar implementation is linearizable.*

PROOF. We define the linearization point of each operation as follows:

- A WaitStep1 linearizes at line 2.
- A WaitStep2 linearizes at line 3 where it reads $spin_p$ is false.
- A NotifyOne linearizes at line 4.
- A NotifyAll linearizes at line 6.

A refinement mapping function simply takes set Q of the generic CondVar implementation as the abstract set Q in the specification. It is easy to see that every WaitStep1, NotifyOne and NotifyAll operations linearizes at its linearization point, by adding or removing threads from the set. When a WaitStep2 operation by thread p linearizes, by Lemma 2, we have $p \notin Q$, and hence, the operation always returns a correct response. $\square$

Algorithm 3: Data Types and Variables

```
type QueueNode
  $sem$    : sem_t        // reference to a semaphore
  $next$   : QueueNode    // next entry in queue
thread local variables
  $my\_node$  : QueueNode    // reference to a queue node
shared variables
  $head$   : QueueNode    // reference to head of queue
  $tail$   : QueueNode    // reference to tail of queue
```

Algorithm 4: The Wait algorithm, using continuation passing

```
procedure Wait($Sync, Cont$)
1   $my\_node.next \leftarrow$ nil
    // Insert thread's semaphore into CondVar's queue
2   BeginTransaction ()
3       if $tail =$ nil and $head =$ nil then
4           $head \leftarrow tail \leftarrow my\_node$
5       else
6           $tail.next \leftarrow my\_node$
7           $tail \leftarrow my\_node$
8   EndTransaction()
    // Break atomicity by completing enclosing sync block
9   EndSyncBlock($Sync$)
    // Wait for a notify
10  SemWait($my\_node.sem$)
    // Execute continuation using same sync mechanism
11  BeginSyncBlock($Sync$)
12  Execute($Cont$)
13  EndSyncBlock($Sync$)
```

3 Design and Implementation

We now present a complete implementation of condition variables that is compatible with both locks and transactions. The implementation satisfies the specification from Algorithm 2.

3.1 Data Structures

A practical condition variable implementation must ensure threads yield the CPU when they are waiting for a notification, and that they wake quickly in response to Notify (i.e., calling `yield` in a loop is insufficient; the thread must be explicitly woken up). Typically, this is achieved by implementing condition variables as OS objects. In contrast we represent each condition variable as a queue in user-space, with the per-thread $spin_p$ flags implemented as binary semaphores. The queue stores references to individual threads' semaphores. By initializing the semaphores to 0, we can remove line 1 from Algorithm 2 and implement line 3 as as SemWait(sem_p). The instances of $spin_x \leftarrow false$ on lines 5 and 7 can each be replaced with SemPost(sem_x). Algorithm 3 presents the data structures for this implementation.

3.2 Algorithm Description

In the interest of generality, we assume a continuation-passing style of execution. The call to Wait thus takes two parameters: an abstract description of the synchronization context, and the continuation to execute after the thread resumes execution. As we discuss later in this section, our implementation can be adapted to other styles with little effort. Algorithm 4 presents an implementation of Wait using this interface, and Algorithm 5 presents NotifyOne.

Algorithm 5: The NotifyOne algorithm

```
procedure NOTIFYONE()
1   BEGINTRANSACTION ()
        // If queue not empty, dequeue head element
2       $sn \leftarrow head$
3       if $sn = $ nil then
4           return
5       if $head = tail$ then
6           $head \leftarrow tail \leftarrow$ nil
7       else
8           $head \leftarrow head.next$
        // Wake the thread when exiting from outermost txn
9       REGISTERHANDLER ($\{$SEMPOST$(sn.sem)\}$)
10  ENDTRANSACTION ()
```

We expect WAIT to be called from an active synchronization context. That is, *Sync* should refer to a mutual exclusion lock that is held by the caller, or a transaction that is being executed by the caller. (We defer discussion of nested critical sections until Section 4). The thread uses a transaction to enqueue its unique node into the CondVar's queue. The use of transactions provides generality and safety: since both WAIT and NOTIFYONE use transactions to access the queue, both methods can be called from any combination of lock-based code, transactional code, and even unsynchronized code, without risking data races on the queue. Strictly speaking, if the CondVar methods are always called from the same type of synchronization context (locks or transactions), this inner transaction is not necessary.

Once the thread has enqueued its semaphore, it then completes its caller's synchronization block, by either releasing the lock or committing the transaction. At this point, we know that descheduling of the caller cannot lead to deadlock: it does not hold resources that are required by another thread. Thus it is safe for the thread to wait on its semaphore. Once the semaphore is signaled, the thread will awake, and execute the continuation (*Cont*) in a synchronized manner, in keeping with the synchronization description present in *Sync*. In comparison to Algorithm 2, we see that the only changes are (a) introducing a synchronization context, and (b) replacing spin-waiting on per-thread flags with the use of per-thread semaphores. Note, too, that by explicitly ending one synchronization context and then instantiating another, we can be sure that there is no active hardware or software transaction at the time of the call to SEMWAIT(sem). Without this guarantee, hardware transactions would abort, due to the system call.

The behavior of NOTIFYONE is simple: using a transaction, the caller removes exactly one element from a nonempty queue, and schedules a signal operation on that element's semaphore. As with WAIT, the use of a transaction ensures race freedom even in the case of naked notifies (i.e., when NOTIFYONE is called from an unsynchronized context). One subtlety is that we use an "onCommit" handler to schedule the semaphore signal to occur when the transaction commits. When NOTIFYONE is called while a lock is held, or from an unsynchronized context, the signal will happen immediately after line 9 completes. However, if NOTIFYONE is called from a transaction, then a waiting thread will not be woken until the notifier's outermost transaction commits. From the perspective of Mesa-style semantics, there is no harm in this approach; the wake-up operation can delay. By delaying the operation, we can be sure that (a) there is no wake-up caused by a transaction that ultimately does not commit, and (b) there is no attempt to call SEMPOST(sem) from an active hardware transactional context. As with WAIT, such a call would cause the hardware transaction to abort and restart in software mode. Note, too, that the current GCC TM implementation maintains the necessary data structures to allow a hardware transaction to store onCommit handlers and run them after transaction commit.

3.3 Supporting NotifyAll

Adding NOTIFYALL support is relatively straightforward. We need only dequeue all elements from the CondVar's queue, and then schedule each element's semaphore to be signaled. An implementation appears in Algorithm 6.

Algorithm 6: The NotifyAll algorithm

```
procedure NOTIFYALL()
1   BEGINTRANSACTION ()
        // If queue not empty, dequeue all elements
2       $sn \leftarrow head$
3       if $sn = $ nil then
4           return
5       $head \leftarrow tail \leftarrow$ nil
        // Wake all threads when exiting from outermost txn
6       while $sn \neq$ nil do
7           REGISTERHANDLER ($\{$SEMPOST$(sn.sem)\}$)
8           $sn \leftarrow sn.next$
9   ENDTRANSACTION ()
```

The principal burden of this algorithm is to ensure that accesses to a queue node's $next$ pointer do not race with nontransactional accesses on line 1 of WAIT. Note that there is a form of privatization taking place: once a thread's node is removed from the queue, there should be no references to the node from any thread other than the node's owner; otherwise, the unsynchronized write on line 1 of WAIT would not be correct.

NOTIFYALL guarantees that all accesses to $next$ fields are performed within a transaction. In order for these elements to be accessible to the thread, the element owner must have committed a transaction on line 8 of WAIT, and there cannot have been an intervening NOTIFYONE or NOTIFYALL that removed that thread's node from the queue. Thus it is impossible for the waiting thread to have reached line 11 of WAIT, and no race is possible.

3.4 Algorithm Properties

Our implementation provides the following properties and guarantees to programmers:

Yielding The history of monitors extends back to a time when uniprocessors were prevalent. Even with multicore CPUs, multiprogramming and oversubscription of threads necessitate support for descheduling a waiting thread, and running another thread on the same CPU. Any practical implementation of condition variables must ensure that upon reaching line 3 of Algorithm 2, the calling thread is put to sleep, and also that the delay between when NOTIFYONE is called, and when the corresponding thread wakes, is minimal. Clearly, our generic algorithm fails in this regard, as it uses a busy wait loop. Even replacing the busy wait with a call to *sched_yield* would not suffice, as it would not guarantee quick wake-up after a notification. However, our use of semaphores addresses this requirement.

Deterministic Wake-Up Semantics In Hoare's work [9], the set associated with a condition variable is explicitly stated to be a queue. Furthermore, a NOTIFYONE operation (there was no NOTIFYALL) was required to be performed while holding a lock, and immediately transferred the lock to the thread at the head of the queue. Mesa, on the other hand, delayed NOTIFY until the notifier reached the end of the critical section. This delay, and the absence of an explicit hand-off of the lock, allowed for higher performance at the cost of weaker semantics: when thread a notified thread b, there was no mechanism to prevent some other thread c from entering the monitor after a completed but before b resumed. This property is shared by our implementation: When a NOTIFYONE pairs with a WAIT operation, attempts to enter critical sections or run irrevocable transactions in other threads can cause the appearance of an unbounded delay after line 10 of WAIT.

On the other hand, the use of a generic set, rather than a queue, matches the C++11 and pthread specifications. Thus it is possible that NOTIFYONE may wake any waiting thread, without regard for which thread began waiting first. Scherer and Scott argued that both stack (LIFO) and queue (FIFO) semantics are sometimes advantageous [18], particularly with respect to caching. Our relaxed (i.e., Mesa) semantics for condition variables, coupled with the user-space implementation, allow for arbitrary thread selection policies, with FIFO as the default. Indeed, since the set is in user-space, it is possible to provide a NOTIFYBEST operation, which traverses the set and selects the best thread to wake (possibly using priority, or an additional parameter provided to the WAIT operation to describe the predicate upon which each thread is waiting).

Spurious Wake-Ups Our specification does not allow spurious wake-ups. That is, a call to WAIT cannot return unless it matches with exactly one notifying operation. This is not expensive in our algorithm, because the act of putting a thread to sleep need not be atomic with the linearization of its upper half, and thus we do not require custom OS support.

In contrast, both the C++11 and pthread specifications allow for a call to WAIT to return even in the absence of a subsequent NOTIFYONE or NOTIFYALL. This relaxation of the specification appears to be a consequence of how operating systems implement condition variables, and in particular how they respond to interrupts that arrive while a call to WAIT is in the midst of transitioning between user-space and kernel execution. In these systems, it must be assumed that *any* call to WAIT may simply return, without a matching NOTIFYONE or NOTIFYALL. Thus even in programs whose logic prevents oblivious wake-ups (see below), WAIT should be called from a loop in order to detect spurious wake-ups. In contrast, our specification does not allow such spurious returns from the WAIT method.

Oblivious Wake-Ups The addition of NOTIFYALL to monitors arose from a common usage pattern in which several threads wait on different predicates, but use the same condition variable. Since traditional condition variables maintain the set of waiting threads in the operating system, it is not possible for NOTIFYONE to know which thread to wake. Consequently, a NOTIFYONE might wake the “wrong” thread, which then must call NOTIFYONE before putting itself back to sleep. The solution, NOTIFYALL, wakes all threads sleeping on a condition variable. When more than one thread is sleeping, we refer to these as “oblivious” wake-ups, since they wake up all of a CondVar’s waiting threads, regardless of whether the predicates upon which they depend have been satisfied.

The possibility of spurious wake-ups necessitates that all threads double-check program data upon return from WAIT, so allowing oblivious wakeups for legacy condition variables imposes no added burden on the programmer in the general case. Given the absence of spurious wake-ups, our implementation only requires double-checking after a WAIT for specific patterns. For example, such checks are not required for single-producer/single-consumer patterns. Note that detecting oblivious wakeups may require the continuation to recursively call WAIT if the caller’s desired predicate does not hold.

4 Implementation and Usage

Up to this point, we have avoided details related to the synchronization contexts within which our CondVar objects may be accessed. Clearly the use of transactions to protect shared data suffices to allow the WAIT, NOTIFYONE, and NOTIFYALL methods to be called from any context. In practice, however, undesirable orderings between waiting threads and notifying threads require that WAIT only be called in conjunction with synchronization mechanisms, such as lock-based critical sections and transactions.

4.1 Lock-Based Critical Sections

When the synchronization context ($Sync$ in Algorithm 4) represents a single lock, there are two manners in which our CondVars can be used in place of traditional condition variables. In the first case, if WAIT is the last instruction within a critical section, then the implementation will observe a null continuation. In this case, we can elide lines 11–13 of the WAIT algorithm: the thread adds itself to the set, calls ENDSYNCBLOCK() to release its lock, and then calls SEMWAIT(). Upon wakeup, the thread need not re-acquire the lock, run an empty continuation, and release the lock. For uses that fit this pattern, this optimization avoids a lock acquire/release pair, decreasing latency and reducing contention on the lock.

In legacy code, rather than create explicit continuations, an alternative is to remove lines 12–13 from Algorithm 4, and remove the $Cont$ parameter. With this interface, the behavior of our CondVar, when called from a lock-based critical section, is indistinguishable from pthread or C++11 condition variables: the caller executes the continuation upon returning from the call to WAIT, and does so using the same synchronization mechanism (the same lock) as was in use at the time of the call to WAIT.

This situation can be generalized to a nested monitor environment, in which several locks are held. If $Sync$ stores references to all locks held at the time of the call to WAIT, then all locks can be released (in any order) on line 9, and then can be re-acquired (presumably in order from outermost to innermost [24]) on line 11, after which the function returns. As with the single-lock setting, the use of a continuation is not required.

4.2 Software Transactional Contexts

The main benefit of a continuation-based API is for calls to WAIT from a software transaction. This is due to the way that stack variables are managed within a transactional context.

To illustrate the subtleties of using a non-continuation-based interface with software transactions, consider the code in Algorithm 7. In this code, a thread initializes function-local variable `outer` on line 1, then begins a transaction. The transaction uses `outer`, along with transaction-local variable `inner` to create the parameters to MAYINVOKEWAIT. The MAYINVOKEWAIT function potentially calls WAIT. Upon return, both `outer` and `inner` are used within the continuation to compute the final value of `outer`, which is then used outside of the transaction.

In the absence of condition variables, BEGINTRANSACTION creates a new lexical scope, which is closed by ENDTRANSACTION. The compiler uses a lightweight instrumentation to checkpoint any stack variables that are both live-in and live-out to the transaction, and which are modified by the transaction. In our example, `outer`

Algorithm 7: An example illustrating the subtleties of mid-transaction calls to *cond_wait*.

```
procedure EXAMPLE($param$)
1   stackvar $outer \leftarrow$ F1($param$)
2   BEGINTRANSACTION ()
3       txnvar $inner \leftarrow$ F1($outer$)
4       $outer \leftarrow$ F1($outer$)
5       $inner \leftarrow$ F2($outer, inner$)
6       MAYINVOKEWAIT($outer, inner$)
7       $outer \leftarrow$ F1($outer$)
8       $inner \leftarrow$ F1($inner$) // Abort happens here
9       $outer \leftarrow$ F2($outer, inner$)
10  ENDTRANSACTION ()
11  F1($outer$)
```

is subject to this checkpointing. Since line 7 is dominated by line 4, `outer` need only be checkpointed once, on line 4.

Committing a transaction early (Algorithm 4, line 9) does not create a substantial burden on the STM: If the STM uses undo logging, then the commit discards its checkpoint; no updates to memory are performed. If the STM uses redo logging, the commit must write-back its updates to shared memory. If the address of `inner` had been passed to F2(), in which case aliasing might result in updates to `inner` appearing in the redo log, then ENDTRANSACTION must update `inner`. If the STM assumed lexically scoped transactions, such an update would be to an invalid stack frame, and would be skipped. With early commits, the stack frame storing `inner` is valid at the time of the early commit. Thus we require a small modification to the ENDTRANSACTION function for redo-log STM systems, so that WAIT can end a transaction via ENDSYNCBLOCK() (line 9). Since the continuation is scheduled in a lexically scoped transaction (lines 11-13), there are no further concerns.

Now suppose that the continuation style is replaced with a more traditional approach. Lines 12-13 of Algorithm 4 would be removed, so that after waking, the thread would begin a transactional context and return. The first concern is somewhat esoteric: if the stack frame for WAIT is on a different virtual page than the stack frame for EXAMPLE, then after WAIT returns, the OS must not invalidate that page, or else an abort on lines 7-9 of example would result in an attempt to restore a stack frame on an invalid page. To the best of our knowledge, modern operating systems do not reclaim stack pages in this manner.

The greater challenge is that if an abort occurs on line 9, after `inner` and `outer` have been modified, then when the transaction restarts (on line 11 of WAIT), the variables must be restored to their state at the time of the call to MAYINVOKEWAIT. This requires two new forms of stack checkpointing.

First, observe that if the compiler is unaware of the possibility of WAIT causing an early commit, then the compiler will not checkpoint `inner`: it is local to the transaction. Thus the implicit BEGINTRANSACTION on line 11 of WAIT must create a checkpoint of the stack frames between it and the outermost transaction involved in the *Sync* context. This requires a small change to BEGINTRANSACTION in the common case: it must store the address of the bottom of its stack frame. Then, when line 11 of WAIT calls BEGINTRANSACTION, it must also copy all stack contents between the stored address and the currently active stack frame. Upon an abort, these contents can be restored, thereby resetting `inner` to its value at the time of the call to MAYINVOKEWAIT().

The second technique is to checkpoint variables, like `outer`, that are neither shared nor transaction-local. With our above modifications to BEGINTRANSACTION, line 2 of Algorithm 7 may or may not believe that `outer` is within its stack frame (the outcome is dependent on the compiler optimization level). However, if `outer` is modified within the transaction and before the call to MAYINVOKEWAIT, then there will be a checkpoint record in the transaction's undo log. Within WAIT, an ad-hoc checkpoint must be created at some point between lines 9 and 11: Before resetting the undo log of the transaction that began on line 2 of EXAMPLE(), we must traverse the undo log to find any address a that is (a) on the caller's stack, (b) not transaction-local, and (c) not shared.[1] Every such address is dereferenced so that the address and its current value can be stored in the ad-hoc checkpoint. If the transaction then aborts (e.g., on line 9 of EXAMPLE), then the transaction can restore the ad-hoc checkpoint before resuming, so that, e.g., `outer` will hold the value that was set on line 4.

4.3 Other Considerations

The complexity of the above mechanism is offset by a number of special cases in which it is not needed. First and foremost, we observe that hardware transactions need not be lexically scoped [10, 11]. Thus when *Sync* is a hardware transaction, the checkpoints are not required. Similarly, if it is known that the code between WAIT and the ultimate commit of the outer transaction does not include self-abort, then it is possible to run the continuation irrevocably [22,23], in which case checkpointing is not necessary, and rollback is not possible. Of course, for long-running continuations, such an approach could greatly impede scalability.

Second, we observe that as with lock-based critical sections, empty continuations avoid this complexity. If lines 7-8 of EXAMPLE were no-ops, then instead of checkpointing and starting a new transaction, line 11 of WAIT could set a flag. WAIT would then return, and the call to ENDTRANSACTION on line 9 of EXAMPLE would immediately return if the flag was set. Another approach when the continuation is empty is to remove line 9 of WAIT, schedule line 10 via REGISTERHANDLER, and then return. In this setting, the empty continuation implies that control flow will return directly to the ENDTRANSACTION in EXAMPLE, which will commit and call its handlers; this, in turn, will call SEMWAIT().

Lastly, we note that unlike lock-based critical sections, transactions are likely to be nested, such that the *Sync* context may represent the flat nesting of several transactions. In STM, this is typically represented by a counter, and thus when WAIT begins a new transactional context on line 11, it must set the counter appropriately. In hardware TM, it may be necessary to call the hardware transaction begin function repeatedly, in order to re-create the appropriate nesting level. Depending on the implementation, this might require a small extension to the software runtime for hardware transactions, to track the nesting depth.

5 Evaluation

In this section, we evaluate the performance of our implementation of transaction-safe condition variables. We seek to answer two questions, one quantitative and the other qualitative:

- What is the overhead of these condition variables, versus pthread condition variables, in lock-based code?
- What anomalies arise when using these condition variables from transactions?

[1]Verifying this last property is not difficult for most STM algorithms, given the metadata they keep about locations they have read and written.

5.1 Experimental Platforms

We performed experiments using two machines. "Westmere" indicates a 6-core/12-thread Intel Xeon X5650 CPU running at 2.67GHz; "Haswell" indicates a 4-core/8-thread Intel Core i7-4770 CPU running at 3.40GHz. Both machines ran Ubuntu 13.04 with kernel version 3.8.0. All benchmarks were compiled using an experimental version of GCC, version 4.9.0. The compiler was configured to use its ml_wt software TM algorithm on Westmere, and to use its HTM algorithm on Haswell. All code was compiled at -O3 optimization level, and experiments are the average of five trials. Variance was uniformly low.

5.2 Benchmarks

We evaluated the performance of our condition variable library using eight benchmarks from the PARSEC benchmark suite [2]: facesim, ferret, fluidanimate, streamcluster, bodytrack, x264, raytrace and dedup. Of the 16 benchmarks in PARSEC, three are "network" versions of other benchmarks within PARSEC, and five benchmarks (blackscholes, freqmine, swaptions, vips and canneal) do not use condition variables. We evaluate the remaining eight benchmarks, which are expected to be representative of the general conditional synchronization patterns that are used widely in current shared-memory multi-threaded programs:

- **facesim** computes the animation of an input modeled face by simulating its underlying physics. It uses condition variables to implement a dynamic and load-balanced task queue that is shared by a group of working threads. The main program adds tasks to each task queue and waits for the completion of these tasks by the working threads.
- **ferret** is a benchmark for content-based similarity search. To process input data (i.e., images), ferret uses a pipeline that contains 6 stages, each stage containing a thread pool and a job queue. From the perspective of condition synchronization, this benchmark represents a pipelined multi-producer, multi-consumer problem.
- **fluidanimate** simulates incompressible fluid for interactive animation. Condition synchronization is only used to implement a barrier, in place of `pthread_barrier`.
- **streamcluster** solves an online clustering problem. It uses condition variables to implement a barrier, and also employs condition variables to allow a master thread to distribute work in a master/slaves pattern.
- **bodytrack** is a computer vision application that can track the 3-D pose of a human body through a series of images. Condition variables are used to implement three synchronization facilities: a barrier, a multi-threaded synchronization queue, and a persistent thread pool.
- **x264** is an H.264/AVC video encoder in which each thread encodes one frame at a time and all threads work in parallel. The use of condition variables in x264 is to coordinate the threads in the encoding process and threads waiting for reference frames.
- **raytrace** is a renderer that generates animated 3D scenes. Multiple threads use a multi-threaded task queue, which employs condition variables.
- **dedup** compresses data streams via a 5-stage pipeline, where each stage employs a queue. Condition variables are used in two settings: the per-stage queues, and a coordination mechanism between worker threads and the (serial) output thread.

Of these benchmarks, fluidanimate, streamcluster and bodytrack use condition variables in place of pthread barriers. Strictly speaking, these uses are not necessary. We measure the condition variable-based barrier nonetheless. All benchmarks except facesim and fluidanimate can run with any number of threads. facesim can only run with a number of threads designated by its input file, while fluidanimate can only run with a power-of-2 number of threads. All benchmarks were tested with their largest available inputs: for facesim, we use input "sim_dev", for others, we use input "native".

Benchmark	Total Transactions	CondVar Transactions	Refactored Continuations
facesim	9	2	0
ferret	3	2	2
fluidanimate	9	2 (2)	2 (2)
streamcluster	7	3 (2)	2 (2)
bodytrack	9	2 (1)	2 (1)
x264	4	1	0
raytrace	14	4 (1)	0
dedup	10	3	3
TOTAL	65	19 (6)	11 (5)

Table 1: Synchronization characteristics of PARSEC source code. Numbers in parenthesis indicate calls to `cond_wait` used in the barrier implementation.

5.3 Software Systems Compared

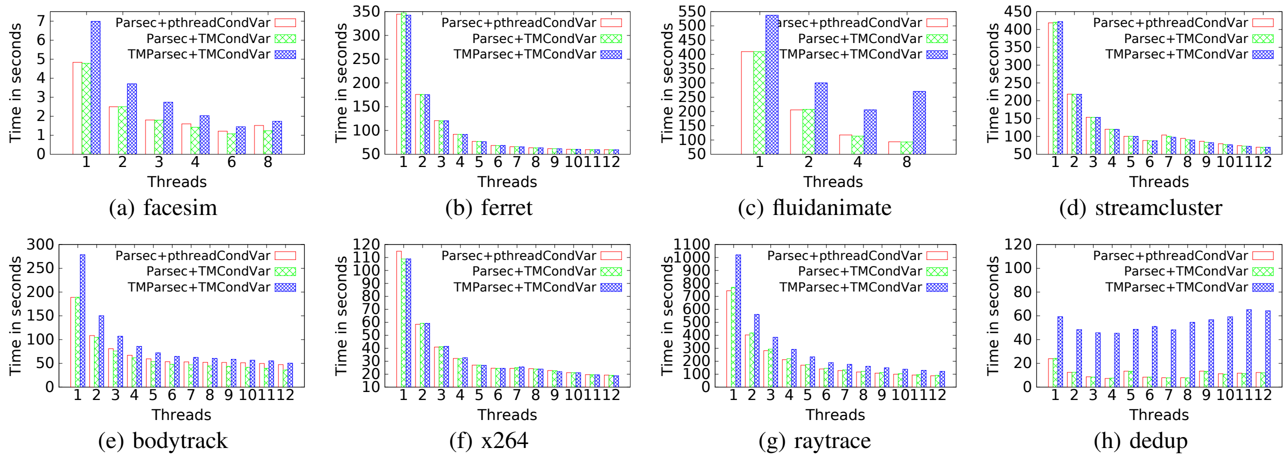
We compare three alternatives. Parsec+pthreadCondVar, the baseline, uses pthread locks to protect critical sections in PARSEC, and pthread condition variables for condition synchronization. Second, we evaluate our transaction-friendly condition variables in a setting where PARSEC still uses pthread locks (Parsec+TMCondVar). This implementation uses transactions internally to protect the condition variables' internal queues, and thus is compatible with code that calls CondVar methods from lock-based, transactional, or unsynchronized contexts. By comparing Parsec+TMCondVar to Parsec+pthreadCondVar, we can determine whether our mechanism offers competitive performance for lock-based code relative to the current state-of-the-art, and thus implicitly whether the use of transactions in the implementation creates unacceptable overheads. Note that on the "Westmere" machine, the internal implementation uses software transactions, but on "Haswell" the internal implementation employs hardware TM.

Our third comparison point, TMParsec+TMCondVar, replaces all locks in the eight PARSEC benchmarks with transactions, and uses our transaction-friendly implementation of condition variables. To ensure a consistent interface, we opted not to use the continuation passing style, and instead to use manual refactoring to split transactions at the point of a WAIT (see Section 4). Table 1 shows that this effort required refactoring of very few critical sections.

5.4 Performance

Figures 1 and 2 show the performance of PARSEC benchmarks on the Westmere and Haswell machines. Figure 3 presents the geometric mean speedup of each software system, versus the baseline (pthread) implementation of condition variables.

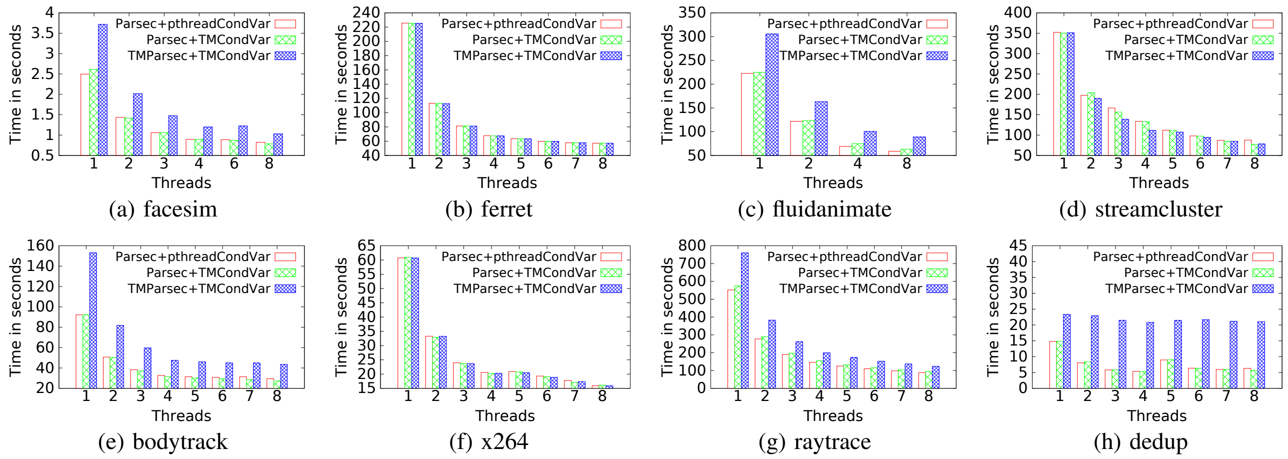
Cost of Transaction-Friendly Condition Variables Comparing Parsec+pthreadCondVar to Parsec+TMCondVar reveals the overheads that come from using semaphores and transactions to implement condition variables, instead of an OS mechanism. On both Westmere (software TM) and Haswell (hardware TM), the cost of our implementation is negligible: at times, our mechanism outperforms pthread condition variables by a small margin, and otherwise it is outperformed by a similarly small margin. Since all transactions are small (fewer than 10 locations), neither artificial conflicts (STM) or artificial fallbacks to a single lock (HTM) occur.



(a) facesim (b) ferret (c) fluidanimate (d) streamcluster

(e) bodytrack (f) x264 (g) raytrace (h) dedup

Figure 1: Westmere performance



(a) facesim (b) ferret (c) fluidanimate (d) streamcluster

(e) bodytrack (f) x264 (g) raytrace (h) dedup

Figure 2: Haswell performance

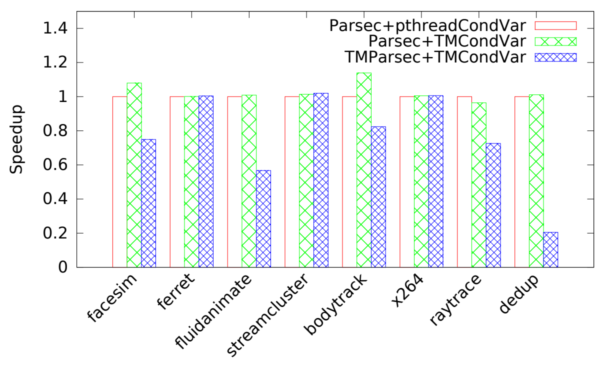
Transactionalized PARSEC TMParsec+TMCondVar represents the first time that these PARSEC benchmarks have been transactionalized on a real-world system, since no prior work has provided support for condition variables. The benchmark performance roughly falls into three categories. First, on streamcluster, ferret, and x264, performance is roughly equivalent to baseline when all locks are replaced with transactions. Second, in facesim, fluidanimate, bodytrack, and raytrace, performance shows the same tendencies as for lock-based code, but with higher overhead. This should not be a surprise: naive transactionalization should not be expected to outperform carefully-tuned lock-based code, especially for large or conflicting transactions. We leave as future work investigation into the exact causes of these slowdowns, and remedies.

The final benchmark, dedup, exhibits virtually no scaling. In dedup, there is a critical section that performs I/O within a relaxed transaction. These relaxed transactions cannot run in parallel with any other transactions, and thus during I/O, there is no concurrency. While this situation has long been expected by the research community, dedup provides a concrete data point.

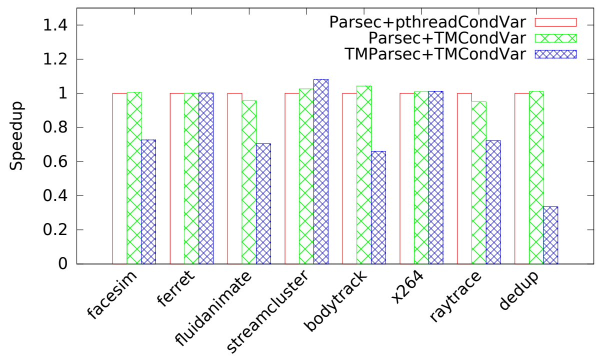
Summary Figure 3 summarizes these results: our transaction-friendly condition variables impose negligible performance degradation across all benchmarks, on both machines. The best case speedup, on bodytrack, reaches 120% on the Westmere system, and 110% on the Haswell system. While there is clearly more work to be done before TM performs as well as locks for PARSEC, the road ahead should be much clearer now that it is possible to transactionalize these 8 benchmarks. Regarding the TMParsec+TMCondVar bars, we encourage the reader to treat the results as qualitative: at long last, condition variables can be supported when transactionalizing legacy code, and for many condition synchronization patterns, such integration is seamless and does not impair performance. These benchmarks still may require tuning to perform well when using transactions, but such an effort is outside of the scope of this paper.

6 Related Work

Research into condition synchronization mechanisms for transactions covers a wide spectrum. On one side, there are efforts, like ours, to improve the implementation of condition variables and/or provide transaction-safe condition variables. On the other side of the spectrum are efforts to craft new alternative condition synchronization mechanisms, which bear little resemblance to traditional monitors and condition variables. While our work is the first to explore condition synchronization in a manner that is compatible with both commodity hardware TM and the C++ TM specification,



(a) Westmere



(b) Haswell

Figure 3: Geometric mean speedup versus baseline

its relationship with prior work is complex. We highlight salient foundational and related work below.

Our work to replace condition variables with semaphores bears a similarity to efforts undertaken by Birrell [3]. That work attempted to create condition variables for a general-purpose operating system (Win32) using only semaphores. A key consideration was whether it would be possible to implement each condition variable with a constant number of semaphores. Many corner cases arose, which ultimately led to the creation of first-class condition variables in later versions of Win32 operating systems. Birrell's work preceded widespread language-level support for thread-local variables, and thus did not consider the alternative we propose, of using per-thread semaphores instead of per-condition-variable semaphores.

In another closely related project, Dudnik and Swift [6] explored the hardware and OS mechanisms required to make transactions compatible with the Solaris OS's implementation of condition variables. Their work considered a robust hardware TM that had internal support for onCommit handlers, and emphasized compatibility with a legacy implementation of condition variables. As a result, the main considerations dealt with supporting system calls from within a hardware transaction, and making OS mechanisms compatible with transactional synchronization. In contrast, our work moves all but the most bare-bones scheduler interaction into user-space, and does so in a manner that is agnostic to the TM implementation and OS. Our work thus generalizes and simplifies this earliest effort at transaction-safe condition variables.

Smaragdakis et al. proposed punctuated transactions as a means of handling I/O and condition synchronization [20]. A punctuating action commits one transaction; user code can then run non-transactionally, after which the continuation executes in a transaction. The continuation must contain programmer-supplied code to verify or restore any invariants that were violated during the period between the two "halves" of the transaction. This model has largely been ignored by the TM community for its complexity; however, the notion of restoring invariants upon resumption of a continuation is precisely the model used for monitor-based synchronization. Our implementation of WAIT can thus be thought of as a specialization of punctuated transactions, in which the only code between the punctuated halves is a SEMWAIT(sem). It should be straightforward to generalize our implementation to support other forms of punctuation, in a manner that is compatible with both hardware and software transactions.

Harris and Fraser [7], and later the X10 group [5], suggested a Conditional Critical Regions style of synchronization. In this model, the read-only prefix of a transaction determines if a predicate holds, and if not, the transaction aborts and retries. When the predicate holds, the continuation runs in the same context as the predicate test, as a single atomic transaction. To optimize this model, a transaction may make visible the locations it read to compute the predicate, so that it can yield the CPU. The transaction is woken by another thread, after that thread's transaction changes any of the locations upon which the sleeping thread's predicate depends. Harris et al. later extended this approach to a "retry" construct [8], in which transactions can, at any time, determine that a predicate does not hold. At such a point, the transaction may "retry", which rolls back the transaction's effects, makes the transaction's read set visible, and then yields the CPU until some other transaction commits a write to a location that the sleeping transaction had tried to read. Spear et al. [21] later showed that it is possible to manage retry-based read and write set tracking in a manner that is orthogonal to the underlying TM implementation. However, no existing hardware TM systems support "retry": software instrumentation is currently required.

Luchangco and Marathe [15] and Lesani and Palsberg [12] proposed mechanisms for synchronizing transactions via group commit. In these proposals, which can be extended to more closely resemble condition variables [14], an object mediates dependencies between transactions, and certain interactions (such as one transaction waiting and another signaling) create a requirement for the transactions to commit or abort atomically with each other. As with retry, this mechanism is not compatible with hardware TM: since current hardware TM proposals implement one-phase commit, they cannot ensure that two transactions commit or abort together [13]. As a result, coordinating transactions must execute in software on modern commodity hardware TM.

7 Conclusions and Future Work

This work introduces an implementation of the legacy interface for condition variables that is compatible with transactions. We make it possible, for the first time, to replace locks with transactions in existing software, even when those locks are used for both mutual exclusion and condition synchronization. In experiments on the PARSEC benchmark suite, we showed that the overhead of our mechanism relative to pthread condition variables is negligible, and that the ability to make condition synchronization compatible with transactions allows the discovery of performance anomalies when transactionalizing highly-tuned lock-based code.

Despite this improvement to the state-of-the-art, there is much work that remains, particularly with regard to programming models. In our work, we focus on lock-based code, and thus we do not need to concern ourselves with the use of the "transaction_cancel" construct. Suppose, however, that following the return of a WAIT, the continuation attempts to cancel itself. In this case, it is not clear what should happen: should the outer scope be canceled, in which case a NOTIFYONE might be lost? Should only the continuation be canceled? Real-world uses of both cancellation and transactional condition synchronization are needed before a preferred approach can be known. Indeed, the best approach might be to use a mechanism like $retry$ instead. In that case, a significant challenge will be to allow uninstrumented hardware transactions to run concurrently with a retrying transaction, and to be able to call retry themselves.

Despite these challenges, we are confident that our work will enable more widespread use of transactions. In particular, our effort to make condition variables compatible with hardware TM and the C++ specification ensures that programmers can transactionalize more legacy code, and write new transactional code that uses familiar programming idioms.

Acknowledgments

We thank Victor Luchangco and Michael Scott for many helpful suggestions during the conduct of this research.

8 References

[1] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft Specification of Transactional Language Constructs for C++, Feb. 2012. Version 1.1, http://justingottschlich.com/tm-specification-for-c-v-1-1/.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2008.

[3] A. Birrell. Implementing Condition Variables with Semaphores. In *Computer Systems*, Monographs in Computer Science, pages 29–37. Springer New York, 2004.

[4] A. Birrell, J. Guttag, J. Horning, and R. Levin. Synchronization Primitives for a Multiprocessor: A Formal Specification. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, Austin, TX, Nov. 1987.

[5] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, Oct. 2005.

[6] P. Dudnik and M. M. Swift. Condition Variables and Transactional Memory: Problem or Opportunity? In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[7] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[8] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[9] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[10] Intel Corp. *Intel Architecture Instruction Set Extensions Programming Reference*, 319433-012a edition, Feb. 2012.

[11] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 45th International Symposium On Microarchitecture*, Vancouver, BC, Canada, Dec. 2012.

[12] M. Lesani and J. Palsberg. Communicating Memory Transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, San Antonio, TX, Feb. 2011.

[13] Y. Liu, S. Diestelhorst, and M. Spear. Delegation and Nesting in Best Effort Hardware Transactional Memory. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburgh, PA, June 2012.

[14] V. Luchangco and V. Marathe. Revisiting Condition Variables and Transactions. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[15] V. Luchangco and V. Marathe. Transaction Communicators: Enabling Cooperation Among Concurrent Transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, San Antonio, TX, Feb. 2011.

[16] M. Ringenburg and D. Grossman. AtomCaml: First-Class Atomicity via Rollback. In *Proceedings of the 10th ACM International Conference on Functional Programming*, Tallinn, Estonia, Sept. 2005.

[17] W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, Mar. 2014.

[18] W. Scherer and M. Scott. Nonblocking Concurrent Data Structures with Condition Synchronization. In *Proceedings of the 18th International Symposium on Distributed Computing*, Amsterdam, The Netherlands, Oct. 2004.

[19] A. Skyrme and N. Rodriguez. From Locks to Transactional Memory: Lessons Learned from Porting a Real-world Application. In *Proceedings of the 8th ACM SIGPLAN Workshop on Transactional Computing*, Houston, TX, Mar. 2013.

[20] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with Isolation and Cooperation. In *Proceedings of the 22nd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Montreal, Quebec, Canada, Oct. 2007.

[21] M. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[22] M. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, Sept. 2008.

[23] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[24] H. Wettstein. The Problem of Nested Monitor Calls Revisited. *SIGOPS Operating Systems Review*, 12(1):19–23, Jan. 1978.