Condition Variables and Transactional Memory: Problem or Opportunity?

Polina Dudnik and Michael M. Swift
University of Wisconsin–Madison
{pdudnik,swift}@cs.wisc.edu

Abstract

Condition variables complement locks by allowing a program to specify the order of execution. They have been in use, largely unchanged, for three decades and have proved to be a robust and scalable synchronization mechanism. However, most transactional memory (TM) systems provide no complementary mechanism. While new synchronization constructs have been proposed for TM, it seems prudent to support condition variables with transactional memory at least until other proposals prove themselves to be as robust and enduring.

We present an examination of condition variables and transactional memory. We show that straightforward conversion of lock-based condition variable mechanisms to transactions may be incorrect for some TM systems. We present two condition variable designs for transactional memory that require little from the underlying TM system yet can express common usages of conditional waiting. Both versions impose minimal requirements on the TM system and neither requires knowledge of how TM is implemented or modifications to the TM system.

To adapt condition variables to transactions, we split the wait operation into two operations, preparing to wait transactionally and completing the wait after commit. In the first design, we defer signaling until after commit to prevent races between signalers and the waiters they wake up. In the second, we signal speculatively before the transaction commits and depend on transactional conflict detection to prevent races. We show that both perform well on lock-based programs from the PARSEC suite converted to transactions.

1. Introduction

Transactional memory (TM) seeks to simplify concurrent programming by removing the need for explicit locking. While TM provides mutual exclusion, other concurrent programming tasks, such as ordering of events, have not been as well supported. For example, transactions alone do not provide a mechanism for producer and consumer threads to signal to each other when work or buffer space is available. Recent analysis of concurrency bugs shows that event ordering is a significant source of concurrency bugs [14].

While many new transactional synchronization mechanisms have been proposed [4, 5, 10, 11, 16, 24], they have not yet been widely used. It seems prudent to support condition variables with transactional memory at least until other proposals prove themselves to be as robust. Since stabilizing on semantics from Mesa [13], condition variables have been used without change for nearly three decades. Despite programmability problems, such as failing to signal a waiting thread, condition variables have proved useful [9] and have efficient implementations.

Transactional memory changes the semantics of both waiting and signaling. For example, most condition variable implementations warn against "naked notifies", or signaling without holding the condition mutex lock [13]. However, signaling within a transaction raises the possibility that the signaling transaction may abort, so the condition it signaled never occurs. Furthermore, we observe that under some conflict resolution policies a recently woken transaction may cause a signaling thread to abort.

In this paper, we investigate how to use condition variables with transactions. We identify substantial changes in the use/implementation of condition variables.

Waiting. We observe that a regular condition variable drops a lock and waits on a queue to be signaled. This must be atomic, so that an intervening signaler cannot acquire the lock, change the condition, and signal, before the waiter waits on the queue. With transactions, this may not be possible. Waiting on a queue is a kernel operation in some operating systems, such as Linux and Windows, and most current transactional memory systems apply only to user-level code.

We propose splitting the `wait` operation into two operations: *prepare wait* and *complete wait*. The prepare wait statement, executed within a transaction, ensures that the following complete wait outside the transaction will not miss any concurrent signals. While splitting the wait function was previously done in AtomCaml [21], we provide an implementation suitable for C-like languages that builds only on common OS primitives for waiting.

Signal/broadcast. We observe that a signaling or broadcasting transaction may execute concurrently with a transaction that it woke up. This can lead to lost wake-ups if the waking transaction completes before the signaling transaction. The waking transaction may view shared state from before the signaling transaction, and hence before the logical condition it awaits is visibly changed.

We present two solutions to this problem with different requirements on the underlying TM system. The first, *deferred-signal*, delays signaling until after the transaction commits. This raises the cost of signal/broadcast compared to signaling from within the transaction and requires support for deferring actions until the transaction commits. The second solution, *speculative-signal* executes a signal/broadcast from within the transaction. This design requires the ability to escape from the transaction temporarily to call into the kernel. Speculative signals rely on transactional conflict detection to guarantee that if the signaling transaction does not abort then any thread it wakes up will observe its state changes. Unlike most implementations of condition variables, both designs can be implemented without knowledge of or changes to the transactional memory implementation.

We present performance results from simulation with the LogTM-SE TM system [28] showing that our implementation performs well. We demonstrate its utility by converting one program from the libMicro test suite [25] and two programs from the PARSEC benchmark suite to use transactions [1]. These tests demonstrate that transactional condition variables can perform as well as or better than lock-based condition variables.

In the next section, we describe the semantics of condition variables. Following that in Section 3 we discuss alternative conditional synchronization mechanisms. In Section 4 we discuss the design considerations for a TM implementation of condition variables. We present our two approaches in Sections 5 and 6. We follow that with an evaluation and then conclude.

2. Condition variables

Condition variables are commonly accessed through the Pthreads API in C and C++ [19] or through language-based mechanisms, such as Java's wait/notify methods [8]. As the two implementations are similar, we describe the Pthreads interface and semantics.

A condition variable is a queue of waiting threads and supports three operations: wait, signal and broadcast. Logically, every condition variable is associated with one (or more) Boolean *condition expressions* and a mutex lock. The lock protects a *monitor invariant*, a property of shared state that must be true when the lock is not held. A thread acquires the lock and tests the condition expression. If it is not true, the thread waits for the condition expression to become true by calling *wait* on the condition variable and lock. This atomically: (1) enqueues the thread on the condition variable, (2) releases the lock, and (3) suspends the thread. Thus, the thread must ensure that monitor invariant holds before waiting.

A thread, while holding the lock, may change the variables constituting the condition expression. It should then call *signal* to notify a single waiting thread to resume execution or *broadcast* to wake all waiting threads. Signal and broadcast move waiting threads off the condition variable's queue and queue them on the associated lock. Only after the signaling/broadcasting thread releases the lock may these threads resume execution. However, it is possible that other threads were previously waiting on the lock, so the waiting threads are not guaranteed that the condition expression is true. Rather, returning from wait is considered a *hint* that expression may have become true. After resuming, the thread must test the expression again before proceeding. Hence, threads commonly wait within a while loop that tests the condition expression. Figure 1 shows an example of waiting.

These semantics for condition variables were first implemented in the Mesa language [13] and have remained largely unchanged since then. They provide several benefits:

1. *Constant-time operation.* Signaling and waiting are both independent of the number of waiters, as they only manipulate the head or tail of the queue.
2. *Flexibility.* When multiple condition variables are associated with a single lock (possible in Pthreads but not Java), a signaler can wake specific waiters.
3. *Speed.* As waiters are specifically signaled, it is likely that when they resume the condition will be true.

As evidence of the durability of condition variables, Microsoft recently added support for condition variables to Windows Vista [22] after the difficulty of implementing condition variables on existing primitives was shown [2].

Despite their popularity, condition variables have two limitations. First, the *signal* operation is explicit, so a programmer may forget to call signal when changing shared state. Second, condition variables do not nest within multiple levels of mutex locks [13] because only the inner-most lock is released when waiting. As a result, deadlock may result if the blocked thread can only be woken by code that acquires a lock still held by the waiting thread.

3. Alternative Synchronization Mechanisms

The incumbent conditional synchronization mechanism for transactional memory is the `retry`/`orelse` construct [11]. When a transaction invokes `retry`, it aborts to the outermost level (if there was nesting) and waits until any address it read is written by another transaction. Compared to condition variables, `retry` eliminates the need for signaling, because transactions are restarted automatically if anything changed. The `orelse` statement allows a transaction to try multiple sub-transactions, each of which retry; only if all sub-transactions invoke `retry` does the parent transaction abort and wait. Thus, `orelse` acts like a socket `select` call by allowing a function to wait for one of many sub-transactions to become ready to execute. This enables composition of multiple transactions that wait, which is not possible with condition variables. A more limited version of `retry` in which the programmer specifies memory locations

```
void BeginWrite() {
  pthread_mutex_lock(mutex);
  while (NWriters == 1 || NReaders > 0) {
     ++WaitingWriters;
    pthread_cond_wait(CanWrite,mutex);
    --WaitingWriters;
  }
  NWriters = 1;
  pthread_mutex_unlock(mutex);
}
```

Figure 1. Condition variable code that depends on waking the correct thread. The `WaitingWriters` counter causes readers to wait until a writer proceeds.

to watch has been designed for a proposed hardware TM system [4, 16].

However, we observe that `retry` may not be applicable to all synchronization problems. First, all waiting threads must execute when something they read changes. This effectively turns all signal operations into broadcasts, because the programmer cannot express a preference between threads. Second, `retry` may incur additional performance costs as more threads wait, as there are more threads to check for conflicts. In contrast, condition variables have constant-time execution. Thus, condition variables are still an important synchronization construct that should be supported.

Other transactional memory systems have implemented *conditional critical regions* [12] by specifying a predicate that must be true to execute the subsequent transaction. The thread executes the predicate and, if it is not true, waits until another transaction modifies the memory locations accessed by the predicate [5, 10, 24]. Similar to `retry`, this construct avoids the need for an explicit signal operation at the cost of turning every signal into a broadcast.

Given the interest in automatic synchronization, in which threads are woken without specific instructions from the program, it is perhaps instructive to re-read the discussions that lead to the creation of monitors and condition variables. Hoare, a co-creator of monitors, wrote about using conditional critical regions within operating systems:

> I feel this proposal [conditional critical regions] is not suitable for operating system implementation. My proposed method encourages the programmer to ignore the question of which of several outstanding requests for a resource should be granted [9].

Figure 1, part of a condition variable implementation of a readers/writers lock illustrates this problem. The code uses the `WaitingWriters` counter to block readers until a writer proceeds. With conditional critical regions, waiting occurs without any changes to shared state.

Brinch Hansen, the other co-creator of monitors, identified three additional weaknesses of conditional critical regions [9]:

1. "The resource concept is unreliable." Similar to current debates about strong and weak atomicity, this statement notes that all changes to a variable, whether inside or outside a critical section, must be considered.
2. "The context switching is inefficient." This statement notes that re-evaluating expressions by re-executing them in the waiting thread leads to excessive context switching.
3. "The scheduling mechanism is too restrictive." This statement notes that waiting without side effects (such as aborting before waiting) prevents a thread from indicating the urgency of its request, for example to prioritize some waiters over others. A reader-writer lock, for example, may want to block new readers while a writer is waiting.

Thus, we believe that condition variables will remain a valuable synchronization tool and should be supported by transactional memory systems.

4. Integrating Condition Variables with Transactional Memory

We seek an implementation of condition variables for transactional memory that keeps the same semantics as existing condition variables. Therefore, a transaction must commit before waiting on a condition variable. As a result, nesting condition variables within multiple transactions is not possible.

We have found three important issues that must be resolved to implement condition variables for transactional memory.

Implementation Independence. Most condition variable implementations are tightly coupled to the implementation of locks, to allow the wait operation to atomically enqueue a thread, drop the lock, and wait in the kernel [2]. However, transactional memory has many implementations, in software, hardware, or a mix. Furthermore, TM systems vary widely in how they resolve conflicts, how they manage concurrent access to data by transactions and non-transactional code, and what features they provide.

We seek a design that is compatible with the Pthreads API, applicable across a broad range of TM systems, and requires no modifications to the TM system or knowledge of its implementation.

Waiting. Waiting on a traditional condition variable atomically enqueues the thread on a queue, drops the lock, and waits in the kernel. This ensures that concurrent signal operations do not arrive between testing the condition expression and waiting in the kernel. However, the queue is shared state and may touch per-thread structures, such as control blocks that are also accessed by non-transactional threads. Therefore, enqueuing a thread in a transaction may cause unnecessary conflicts.

As a result, an alternate mechanism besides a queue is needed to prevent lost wake-ups. Furthermore, TM condition variables must split waiting into a `prepare` operation, which takes place transactionally and ensures that subsequent signals are not lost, and a `complete` operation after committing the transaction to enqueue the thread and wait in the kernel.

Signaling. We observe that with mutex locks and condition variables, the order of updating the condition and signaling is irrelevant; as long as the condition expression has been updated before the lock is released, waiting threads will correctly observe the condition to be changed. In a TM system, a waking thread could overlap execution with the signaling thread and test the condition before it is modified, causing either a conflict between the threads or a lost wake up.

Thus, the signal implementation must ensure that waking threads observe the change to the condition expression by a signaling thread.

The signal operation is commonly executed with the lock held to prevent so-called *naked notifies* [13]. With lock-based condition variables, signaling without holding the lock is discouraged for two reasons. First, a naked notify is often accompanied by changing the state variables representing the condition without a lock. When these variables are changed without a lock, wake-ups can be lost because a concurrent waiter may read the state variable, followed by the signaler changing the state variable and signaling, followed by the waiter waiting. Second, in some implementations (including Solaris) the signaler may only move a waiter from the condition variable queue to the mutex queue if it holds the mutex. Otherwise, the waiter must resume execution and attempt to acquire the mutex, causing unnecessary context switches.

With transactions, though, both issues disappear as long as state variables updates occur within the transaction. There is no lock queue, so there is no need to move the waiter to a separate queue when signaling.

Prior work on condition variables for transactional memory implemented Mesa-style condition variables for transactions and observe that the wait operation must be split into two operations to allow the transaction to commit [21] and that the signal operation can be deferred until commit [27]. However, these implementations are each for a specific software TM system and have not been abstracted to work across multiple TM systems with different conflict detection and resolution policies.

In the next two sections we present two designs for TM condition variables with different requirements of the underlying TM platform.

5. Deferred-Signal Condition Variables

Our first design for condition variables works across a wide variety of TM implementations. It places only one demand on the TM system: the ability to specify additional code to execute at commit. This allows the implementation of `signal()` to defer delivery of the signal until the transaction commits.

```
struct tx_cv {
  queue_t waiter_queue;
  spinlock_t queue_lock;
  int counter;
}
```

```
void tx_cv_prepare_wait(tx_cv cv) {
  this_thread.cv_counter = cv.counter;
}
```

```
int tx_cv_complete_wait(tx_cv cv) {
  int result = 0;

/* add self to queue */

  set_waiting_flag(this_thread);
  spin_lock(cv.queue_lock);
  enqueue(cv.queue,this_thread);
  spin_unlock(cv.queue_lock);

/* wait in kernel if no intervening signal */

  if (this_thread.cv_counter == cv.counter) {
    result = kernel_wait();
  }

/* clean up from waiting */

  spin_lock(cv.queue_lock);
  dequeue(cv.queue,this_thread);
  spin_unlock(cv.queue_lock);
  unset_waiting_flag(this_thread);
  return(result);
}
```

Figure 2. Code for waiting on condition variables in transactions with deferred signaling

As illustrated at the top of Figure 2, a transactional condition variable is a queue and a counter; the queue contains the list of waiters, and the counter prevents races between waiting and signaling transactions. As with traditional condition variables, transactions should wait within a while loop, as they may wake up spuriously.

5.1 Waiting

As previously mentioned, waiting is split into two operations: preparing to wait and completing the wait. The `tx_cv_prepare_wait` function, in the middle of Figure 2 is called from transactions when they decide to wait. This function reads a counter; the thread learns of a concurrent signal if the counter value changes before it waits. Note that it modifies no shared state, so multiple threads can concurrently prepare to wait. A thread calls the tx_cv_complete_wait function, shown at the bottom of Figure 2, to actually wait after committing a transaction.

We present the wait function as enqueuing, testing to make sure the counter has not changed, and waiting. This implementation is based on Solaris, where these are the primitive operations used to implement locks and condition variables as well. The kernel_wait returns immediately if the waiting flag set by set_waiting_flag has been cleared by a concurrent signaler. For Linux, which provides the futex abstraction [7], the futex system call performs these three operations at once. This call will return immediately if the value passed (the counter parameter) does not equal the value in memory (the counter in the condition variable). If the counter has not changed, the thread adds itself to a queue and waits. Thus, a concurrent signal will be detected to ensure that the wake-up is not lost.

For simplicity we do not show the code to handle signals or timed waits. In both cases, the waiter must check whether it returned from the kernel because of a condition variable signal or another reason.

In a TM system that provides an explicit transaction commit statement, both prepare- and complete-wait may be wrapped in a single function. For languages with atomic blocks, which are lexically scoped, the code must be reorganized to call complete-wait outside an atomic block. Figure 3 shows the code from Figure 1 reorganized in this way. We have developed macros to do this automatically for transactions where wait is lexically scoped within the atomic block.

```
void BeginWrite()
{
  int DoWait  = 0 ;
  do {
    atomic {
      if (DoWait) {
        --WaitingWriters;
        DoWait = 0;
      }
      if (NWriters == 1 || NReaders > 0) {
        ++WaitingWriters;
        tm_cv_prepare_wait(CanWrite)
        DoWait = 1;
      } else {
        NWriters = 1;
      }
    } // end atomic
    if (DoWait)
      tx_cv_complete_wait(CanWrite);
  } while (DoWait);
}
```

Figure 3. Code from Figure 1 converted to use atomic blocks and transactional condition variables. The DoWait variable records whether a wait is necessary. The code following the wait in the original code is moved to the top of the transaction and does not execute on the first loop iteration.

5.2 Signal/Broadcast

As shown in Figure 4, the signal function increments the counter then wakes the first waiter on the queue (if there is one). Incrementing the counter before checking the queue ensures that a concurrent waiter will detect the change. A concurrent waiting transaction will detect the counter change after enqueuing, so a signal may inadvertently wake two threads - one explicitly pulled from the queue, and another one that was executing when signal was called. This design implements the same Mesa semantics as Pthreads, though, so this is a performance and not a correctness issue. The broadcast function (not shown) is similar except that it removes all threads from the queue and wakes them in the kernel.

This implementation depends on the ability to defer operations until the transaction commits, shown in Figure 4 using the register_commit_action function. A TM system providing commit actions records a list of functions and parameters and executes them in order after the transaction commits and releases isolation [16, 17, 18]. Not all TM systems, though, provide this feature.

```
int tx_cv_signal(tx_cv cv) {
  register_commit_action(tx_cv_signal_commit,
                         cv);
}
```

```
void tx_cv_signal_commit(tx_cv cv) {
  atomic_inc(cv.counter);
  spin_lock(cv.queue_lock);
  if (not_empty(cv.queue) {
     thread = dequeue(cv.queue);
  }
  spin_unlock(cv.queue);
  if (thread) {
     unset_waiting_flag(this_thread);
     kernel_wake(thread);
  }
}
```

Figure 4. Code for deferred signaling of a condition variable from a transaction.

For simple uses of signal, as shown in Figures 5, it suffices to simply relocate the signal to execute after com-

```
1   char get() {            char get() {
2     begin_TX;               begin_TX;
3     while (n == 0)          while (n == 0)
4       wait(not_empty);        wait(not_empty);
5     ch = buf[tail % N];     ch = buf[tail % N];
6     tail++;                 tail++;
7     n--;                    n--;
8     signal(not_full);       end_TX;
9     end_TX;                 signal(not_full);
10    return ch;              return ch;
11  }                       }
```

Figure 5. Code transformed to signal after committing. The signal in line 8 on the left is moved to line 9 on the right. This is possible because the code always signals, and never waits after signaling.

```
void test (int k) {
  if ((state([k + N - 1) % N] != EATING) &&
      (state[k] == HUNGRY) &&
      (state[(k + 1) % N] != EATING)) {
    state[k] = EATING;
    signal(self[k].cv);
  }
}

void pickup (int i) {
  begin_TX;
  state[i] = HUNGRY;
  test(i);
  while (state[i] != EATING)  {
    wait(self[i].cv);
  }
  end_TX:
}
```

Figure 6. Sample code for dining philosophers, which is difficult to transform by deferring signals until the transaction commits because the code signals as part of the `test` function, called before waiting. Code adapted from Operating Systems Concepts [23].

mitting the transaction. However, more complicated uses of condition variables, such as the code shown in Figure 6, may conditionally signal multiple condition variables, or wait rather than commit after signaling. In this case, the call to signal must execute as part of the wait, when the transaction commits. Thus, a commit action can guarantee that the signal will execute no matter where the transaction commits.

5.3 Discussion

The deferred-signal condition variable implementation executes correctly on most TM implementations, because the signal executes *after* changes to the condition state are committed to memory and visible to other threads.

Note that the signal/broadcast operation writes a value that is read transactionally by the `tx_cv_prepare_wait` function. This may have different results depending on whether the TM system is strongly or weakly atomic and how it resolves conflicts between transactions and non-transactional code. With strong atomicity and a transaction-wins conflict resolution policy, a signaler will block until a waiting transaction commits. With strong atomicity and a non-transactional-code-wins policy, a signaler will abort a waiter, which will retry and see a change to the condition expression. Thus, the wake-up will not be lost.

With weak atomicity, a signaler may change the counter while a waiting transaction executes. In this case, the waiting transaction will commit, see the changed counter, and retry. Thus, the counter again ensures that wake-ups are not lost.

Deferring signal until commit may in some cases decrease performance. First, it delays waking threads until after commit, which limits the concurrency between signaling and waiting threads. Second, deferred function calls may be more expensive than in-line calls, because the function and parameter must be recorded and control flow must be interrupted at commit to execute the deferred function. For example, on LogTM-SE [28], commit actions require an extra processor trap to invoke the specified function. On other systems, such as the Intel STM, commit is handled in software and invoking an extra function is relatively cheap [18].

6. Speculative-Signal Condition Variables

The preceding section presented an implementation of condition variables that defers signaling until commit. Here we present a second design of transactional condition variables that differs by signaling speculatively, before committing the transaction.

Signaling before committing the transaction introduces two major differences. First, the signal operation executes as part of a transaction, so it can use transactional operations as part of its implementation. However, signal interacts with the non-transactional wait function and calls into the kernel. It therefore also uses *escape actions* internally to temporarily pause transactional semantics so it can invoke the kernel [17, 29]. The effect of a signal, waking a waiting thread, occurs immediately rather than when the transaction commits. This implementation only works for TM systems that implement escape actions.

Second, when signaling before committing, it is possible that a waiter will awake and execute before the signaler commits. Thus, the implementation must ensure that the waiter will observe a change to the condition expression; otherwise it will go back to sleep and the wake-up will be lost.

Unlike the deferred implementation, speculative condition variables use a flag to indicate whether there are user-level waiters. This flag, shown at the top of Figure 7, allows

```
struct tx_cv {
  queue_t waiter_queue;
  spinlock_t queue_lock;
  int counter;
  int waiters;
}
```

```
void tx_cv_prepare_wait(tx_cv cv) {
  this_thread.cv_counter = cv.counter;
  cv.waiters = 1;
}
```

Figure 7. Code for preparing to wait with speculative-signal condition variables. This implementation adds flag to the condition variable declaration that is set when preparing to wait.

signal and wait to use transactional conflict detection to prevent lost wake-ups as we describe below.

6.1 Waiting

The prepare-wait function, shown in Figure 7, differs from the deferred version by setting a `waiters` flag to indicate that there are waiters. Signalers read and set this flag as well, causing a transactional conflict if they are concurrent. Thus, a waiting transaction will be correctly ordered either before a signaling one, ensuring that it receives the signal; or afterwards, ensuring that it will view any changes made to the condition expression's state.

Writing to a shared value as part of prepare-wait decreases concurrency compared to the deferred version, as concurrent waiters will conflict. However, conflicts on the `waiters` variable can be resolved by stalling all but one transaction, which reduces the cost of the conflict. If a TM system prevents concurrent readers and writers, such as with eager conflict detection, only the first waiter need update the flag. Other waiters can read the flag to ensure they conflict with concurrent signaling threads.

The code for waiting on a condition variable is identical to the deferred-signal version in Figure 2.

6.2 Signal/Broadcast

A transaction signals a condition variable by incrementing the counter, testing for waiters, and if necessary waking up one from the queue. Unlike the deferred-signal version, signal performs the queue and kernel wake up operations in an escape action and the increment of the counter and write to the waiters flag occur within the transaction. The wake up and queue operations are escaped because they access state shared with non-transactional code through the complete-wait operation. In contrast, the counter and flag manipulation occurs within the transaction to prevent lost wake-ups that could occur if a signaler woke a waiter, which then re-read the condition and went back to sleep before the waiter commits.

```
int tx_cv_signal(tx_cv cv) {
  cv.counter++;
  if (cv.waiters) {
    int remaining;
    begin_ESCAPE;

    /* wake up any waiters */

    spin_lock(cv.queue_lock);
    thread = dequeue(cv.queue, &remaining);
    spin_unlock(cv.queue);
    if (thread) {
      unset_waiting_flag(this_thread);
      kernel_wake(thread);
    }
    end_ESCAPE;
    cv.waiters = remaining ? 1 : 0;
  }
}
```

Figure 8. Code for signaling a speculative-signal condition variable. The dequeue operation returns a count of the remaining elements on the list.

Instead, concurrent waiting and signaling threads are guaranteed to conflict: they both write to variables read by the other transaction. Therefore, a concurrent waiter may abort the signaler, leading to a spurious wake-up. Otherwise, the signaler will abort the waiter, which will retry and see the condition has changed. Spurious wake-ups can also occur when a signaling transaction subsequently aborts.

TM systems that are vulnerable to the friendly fire pathology [3] may suffer from livelock if a waiting thread causes the signaling thread to abort; this cycle may repeat persistently. Thus, this implementation requires the TM system to implement a conflict resolution policy that prevents a transaction waking after being signaled from aborting the signaling transaction.

6.3 Discussion

Table 1 shows the major elements of the two alternative designs, why they were chosen, and the requirements for the underlying TM system. The speculative signal implementation of TM condition variables can improve performance compared to deferring signal by removing the deferral mechanism. In addition, it reduces the latency of wake-ups, as woken threads can execute before the transaction commits. However, it cannot be implemented on TM systems that lack non-transactional escape actions, and the cost of transactional conflicts between waiters and signalers may be greater than cost of deferring the signal operation. Hence, the choice between systems comes down to the capabilities of the TM platform and the performance of its operations.

Design	What?	Why?	TM Requirements
Deferred-Signal	• `prepare_wait` only reads data • Defer `signal` until after commit	• Prevent wait-wait conflicts • Prevent signal/wake-up races	• Commit actions
Speculative-Signal	• `prepare_wait` writes `waiters` flag • `signal` writes `waiters` flag • Execute `signal` within transaction	• Prevent signal/wake-up races • Low latency wake-up	• Escape actions • Robust conflict detection

Table 1. Distinct design elements of deferred-signal and speculative-signal condition variables. Common features include splitting wait into prepare-wait and complete-wait, and use of a counter to detect signal/wait races.

7. Evaluation

We evaluate both implementations of condition variables against Pthreads locks and condition variables to answer two questions:

1. Do transactional condition variables exhibit any performance pathologies?
2. Can transactional condition variables perform as well as and replace replace regular condition variables in lock-based programs?

We answer the first question with a microbenchmark that stresses the synchronization system, and the second one by converting two lock-based programs to use transactions.

7.1 Evaluation Platform

We implement both deferred-signal and speculative-signal condition variables for OpenSolaris by extending the libc library with transactional condition variable functions. The new implementation added approximately 400 lines of code, similar in length to the existing condition variable functions.

We evaluated both implementations on the LogTM-SE transactional memory system [28] from the latest public release of the Multifacet GEMS System Simulator [15]. The GEMS simulator integrates accurate functional SPARC processor modeling provided by Virtutech Simics [26] with a detailed memory system simulation model. We simulate a thirty-two-core processor with a shared L2 cache. The key configuration parameters are given in Table 2.

LogTM-SE implements eager version management (also known as *direct update*) and eager conflict detection. We adopt the hybrid conflict resolution with a write-set predictor proposed to resolve pathologies associated with eager version management and eager conflict detection [3].

7.2 Workloads

We use the cascaded_cond microbenchmark from the libMicro version 0.4.0 microbenchmark suite [25] to test the performance of transactional condition variables and the FluidAnimate and StreamCluster programs from the PARSEC shared-memory application suite [1]. Several workloads require a management thread, while others only support powers of two for thread counts. Thus, we could not employ all 32 processors across all workloads. To keep the worker thread count constant, we therefore use sixteen worker threads.

Processor	UltraSPARC-III+
Number of Processors	32
L1 Cache	32 KB, 4-way, 1 cycle
L2 Cache	8 MB, 2-way, 21 cycle
Main Memory	16 GB
Memory Configuration	NUMA (balanced)
Operating System	OpenSolaris release snv_31

Table 2. Key features of the simulated processor.

LibMicro is a portable set of microbenchmarks used by Solaris engineers to measure the performance of various library calls such as conditional variable signal and wait [25]. Threads are arranged in a ring. Each thread has two locks on which it blocks, and threads manipulate the locks belonging to the thread following it in the ring. We converted all locks and condition variables to use transactions.

The PARSEC benchmark suite is intended for x86 machines running Linux and therefore required conversion to SPARC/Solaris. Not all PARSEC benchmarks lend themselves easily to such a conversion and only a subset of these benchmarks are suitable for transactional memory as many benchmarks synchronize infrequently. We selected FluidAnimate and StreamCluster for their use of condition variables and ease of porting. StreamCluster uses condition variables to order the master thread creating work with the slave threads waiting for work. FluidAnimate executes five kernels to simulate an incompressible fluid and uses barriers to delimit simulation stages. We converted the barriers to use conditional signal and wait within transactions. In addition, we converted all other uses of mutex locks to transactions. For both programs, we use the largest input size.

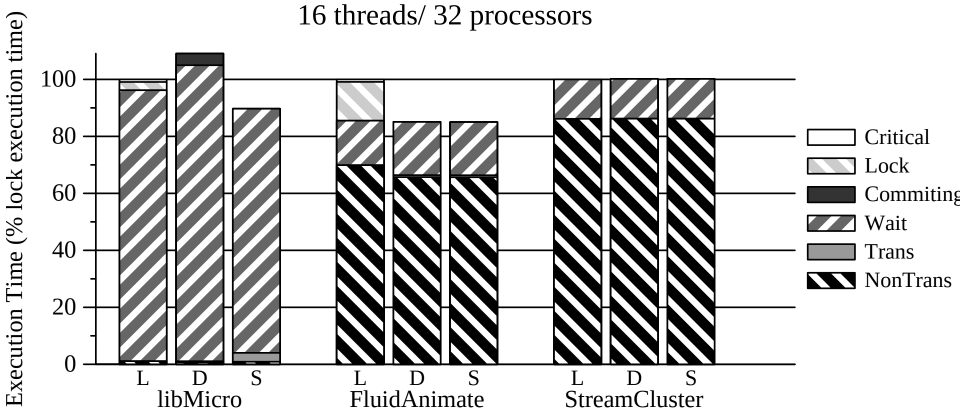

Figure 9. Relative performance of lock and both TM condition variable implementations for libMicro, FluidAnimate, and StreamCluster. The lock version is labeled (L), the speculative-signal version (S),and the deferred-signal (D). Time spent waiting for condition variables is labeled 'Wait'; time outside locks and condition variables is labeled 'NonTrans', and time spent acquiring and releasing locks is labeled 'Lock'

7.3 Results

The results of the simulation are shown in Figure 9, broken down by where each workload spends time in each implementation. We normalize results to the original locking version of each program. Each program exhibits different behavior demonstrating how implementation differences impact performance.

The libMicro workload, shown on the left, does nothing but conditional synchronization. As a result, it spends most of its time waiting and there is little time spent outside waiting or lock functions. We observe two aspects of TM condition variables. First, even under this stressful workload they show no pathologies, indicating that they can be as robust as lock-based condition variables. Second, we observe that the speculative-signal condition variable reduces execution time for this workload, because it spends less time waiting. Signaling early, while the transaction is still executing reduces the wake-up latency and allows waiters to begin executing even before the signaler commits. Thus, speculative-signal condition variables reduces total execution time by 10%. In contrast, the deferred-signal implementation increases execution time by 9% because waiting threads must wait longer to be signaled. In addition, the time spent signaling, which shows up as time spent committing, is longer for deferred-signal because of the cost of trapping to execute a commit action. In comparison, the time to signal with speculative-signal, which is counted as time spent in the transaction (labeled 'Trans'), is short because escape actions add no appreciable overhead.

FluidAnimate and StreamCluster demonstrate the successful use of TM condition variables in larger programs. In both programs, both condition-variable implementations behave identically because little time was spent in conditional synchronization. Thus, transactions and TM condition variables are a viable replacement for locks. Furthermore, we see that neither program spends much time holding a lock in a critical section (labeled 'critical' in the figure) or within transactions (labeled 'Trans'). However, performance for FluidAnimate improves noticeably with transactions. The breakdown in Figure 9 shows why: the locking version spends 15% of its execution time executing locking functions. This cost is reduced by removing lock contention with transactions and because transaction begin/commit executes faster than acquiring and releasing a lock. Previous work has shown that replacing spin locks with TM make critical sections faster by removing expensive lock accesses [6, 20]. Finally, StreamCluster has few critical sections and few condition variable operations. Therefore, all three versions perform the same.

These tests demonstrate that transactional condition variables can perform as well as or better than lock-based condition variables. Furthermore, the PARSEC benchmark results show that they enable replacement of locks with transactions that might not otherwise be possible.

8. Conclusions

Despite the advances in transactional synchronization, condition variables will remain an important mechanism. We find that adapting condition variable implementations to transactions requires consideration of conflicts between concurrent signalers and waiters.

We present two alternative implementations of condition variables for transactional memory, speculative-signal and deferred-signal. They may both be implemented upon an underlying TM system without knowledge of its implementation. The two implementations place different, but modest, requirements on the TM system: deferred-signal requires commit actions to defer signaling until commit, and speculative-signal requires escaping transactions to call into the kernel within a transaction. Both work across a wide variety of TM systems, and their performance differs based on their ability to overlap execution (speculative-signal allows more overlap) and on the relative cost of their underlying operations.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) grant CNS-0720565 and CNS-0551401. Thanks to Mark Hill for valuable feedback on early versions of this paper and Brian Hickman for converting PARSEC programs to use transactions. Swift has a significant financial interest in Microsoft.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT 17*, Oct. 2008.

[2] A. Birrell. Implementing condition variables with semaphores. In *Computer systems theory, technology, and applications*, pages 29–37. Springer Verlag, 2004.

[3] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *ISCA 34*, June 2007.

[4] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *PLDI 2006*, June 2006.

[5] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approachto non-uniform cluster computing. In *OOPSLA 20*, Oct. 2005.

[6] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. In *TRANSACT 3*, Feb. 2008.

[7] U. Drepper. Futexes are tricky. `http://people.redhat.com/drepper/futex.pdf`, Jan. 2008.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] P. B. Hansen. Monitors and concurrent pascal: a personal history. *SIGPLAN Not.*, 28(3):1–35, 1993.

[10] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 18*, Oct. 2003.

[11] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *PPOPP 12*, June 1991.

[12] C. A. R. Hoare. Towards a theory of parallel programming. In *The origin of concurrent programming: from semaphores to remote procedure calls*, pages 231–244. Springer-Verlag New York, Inc., 2002.

[13] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa (summary). In *Proceedings of the seventh ACM symposium on Operating systems principles*, 1979. ACM.

[14] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS 13*, Oct. 2006.

[15] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *Computer Architecture News*, pages 92–99, Sept. 2005.

[16] A. McDonald, J. Chung, B. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA 33*, June 2006.

[17] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS 12*, Oct. 2006.

[18] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for c/c++. In *OOPSLA 23*, June 2008.

[19] B. Nichols, B. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., 1996.

[20] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[21] M. F. Ringenburg and D. Grossman. Atomcaml: First-class atomicity via rollback. In *International Conference on Functional Programming*, Sept 2005.

[22] R. Saccone and A. Taskov. Synchronization primitives new to Windows Vista. *MSDN Magazine*, June 2007.

[23] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, 8th edition*. Wiley Publishing, 2008.

[24] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *OOPSLA 22*, Oct. 2007.

[25] Sun Microsystems. OpenSolaris project: libMicro - portable microbenchmarks. `http://opensolaris.org/os/project/libmicro/`, 2007.

[26] Virtutech Inc. Simics full system simulator. `http://www.simics.com/`.

[27] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *Proceedings of 18th European Conference on Object-Oriented Programming*, June 2004.

[28] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA 13*, Feb. 2007.

[29] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT 1*, June 2006.