Practical Condition Synchronization for Transactional Memory

Chao Wang
Lehigh University
chw412@lehigh.edu

Michael Spear
Lehigh University
spear@cse.lehigh.edu

Abstract

Few transactional memory implementations allow for condition synchronization among transactions. The problems are many, most notably the lack of consensus about a single appropriate linguistic construct, and the lack of mechanisms that are compatible with hardware transactional memory. In this paper, we introduce a broadly useful mechanism for supporting condition synchronization among transactions. Our mechanism supports a number of linguistic constructs for coordinating transactions, and does so without introducing overhead on in-flight hardware transactions. Experiments show that our mechanisms work well, and that the diversity of linguistic constructs allows programmers to chose the technique that is best suited to a particular application.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

Keywords Transactional Memory, Condition Synchronization, Retry, Atomicity, Semaphore

1. Introduction

Transactional Memory (TM) was originally proposed as a hardware mechanism to simplify the creation of nonblocking data structures [16]. It then was embraced as a mechanism for lock elision [24], and has come to be seen today as a full-fledged programming model [1].

TM has a clear and valuable role in increasing concurrency among critical sections, by eliminating the need for locks. When lock-based critical sections are replaced with transactions, those critical sections can run in parallel as long as their memory accesses do not conflict, and will run in a correct sequential order otherwise. However, locks are not the only tool for coordinating threads: many concurrent programs also employ condition variables to suspend thread execution until some precondition is met. The lack of support for some form of condition synchronization presents a challenge to TM adoption and widespread use [29, 35, 36].

Several proposals allow the use of condition variables within transactions [10, 32, 35], but these have not been widely embraced. In contrast, the RETRY mechanism [15] is a popular tool for coordinating transactions in Haskell [34]. The distinction between condition variables and RETRY is significant. Transactional condition variables break atomicity, by committing an in-flight transaction at the point where the transaction must wait, and then running the remainder of the transaction as a new atomic region, after wakeup. In contrast, RETRY casts conditional synchronization as a form of scheduling: RETRY allows the programmer to state that a transaction should not have been started yet, because some dynamically-determined precondition did not hold when the transaction was attempted. When a RETRY is encountered, the transaction's effects are undone and the transaction is not attempted again until some datum read by the most recent attempt is updated by a transaction from another thread.

Because RETRY does not break atomicity, it is composable: When a RETRY by an inner nested transaction causes the outer transaction's effects to be undone, it is as if the outer transaction was never attempted. In contrast, waiting on a condition variable within an inner nested transaction exposes the partial updates of the outer transaction. Thus a programmer can use RETRY within library code, without needing to then perform whole-program analysis to understand the impact of RETRY on outer nested scopes.

Unfortunately, work-conserving RETRY implementations are complex and intimately tied to low-level details of an underlying software TM (STM) implementation [15, 31]. They operate by publishing to a global data structure the list of all metadata associated with locations read by the retrying transaction. This action is done atomically with the retrying transaction undoing its effects. Every subsequent transaction must remember the metadata of every address it updates, so that, during commit, it can compare its write metadata with the suspended transaction's read metadata, and wake the transaction if the intersection is not empty. Today's hardware TM (HTM) systems do not use metadata, nor do they provide a means of seeing a successful transaction's write set. Thus RETRY can only be approximated via a simple

EuroSys '16, April 18-21, 2016, London, United Kingdom

ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2901318.2901342

abort-and-immediate-restart operation. When it is necessary for the retrying transaction to yield the CPU, then as soon as any transaction invokes RETRY, all hardware transactions appear to need to immediately switch to a high-overhead instrumented mode and restart.

This paper introduces DESCHEDULE, a low-level mechanism that addresses two shortcomings of RETRY. First, it is independent of the underlying TM implementation: it uses value-based validation [7, 23] to precisely identify when to re-attempt transactions, rather than STM-specific per-location metadata. Consequently, DESCHEDULE is compatible with all known STM, HTM, and Hybrid TM (HyTM) systems, and is immune to false wakeups due to silent stores. Second, as a low-level mechanism, DESCHEDULE can be used to build a range of conditional synchronization mechanisms. In addition to RETRY, we use it to implement AWAIT [4], which achieves lower space complexity than RETRY by only monitoring one location. We also introduce a new mechanism, WAITPRED, which allows the programmer to use arbitrary predicate functions to control when to re-attempt transactions. In effect, WAITPRED combines the atomicity of RETRY with the explicit and potentially fine-grained guards used in traditional condition variables.

The remainder of this paper is organized as follows. Sections 2 and 3 review conditional synchronization in general, and RETRY in particular. Section 4 presents DESCHEDULE, and Section 5 shows how it can be used to implement RETRY, AWAIT, and WAITPRED. Section 6 contrasts the programmability of these three mechanisms. We implemented the mechanisms within the GCC compiler's TM infrastructure, and in Section 7, we contrast the mechanisms' performance under both software and hardware TM, using microbenchmarks and the PARSEC benchmark suite [3]. Section 8 discusses related work, and Section 9 concludes.

2. A Motivating Example

Listing 1 uses the example of a bounded buffer with transactional condition variables to illustrate some of the key ideas and challenges we address in this paper. The intent of the code is to provide a multi-producer, multi-consumer buffer. When non-transactional code calls the PRODUCE and CONSUME methods, the buffer's behavior is correct. However, a programmer may reason that since there are transactions, it ought to be possible to compose complex atomic behaviors involving the buffer. To this end, suppose the programmer crafts $Produce1Consume2()$ which enqueues one item and dequeues two consecutive items.

Suppose thread T calls $Produce1Consume2()$ when $count$ is 0. Lines 26-30 will execute atomically, resulting in the function setting some shared state, creating a new element, inserting it into the buffer, and extracting the element from the buffer. However, when line 31 is reached, the buffer is empty, and thus line 21 will be reached. To put T to sleep, the outermost transaction will commit, breaking atomicity.

Listing 1: A bounded buffer example using transactions and condition variables with Mesa semantics

```
Shared Fields:
  buf        : Array     // stores the elements in the buffer
  cap        : Integer   // the size of the array
  count      : Integer   // # elements in the array
  nextprod   : Integer   // destination index for next Produce()
  nextcons   : Integer   // source index for next Consume()
  notempty   : CondVar   // condition variable for consumers
  notfull    : CondVar   // condition variable for producers

Internal Methods:
function Full()
1   return count = cap
function Empty()
2   return count = 0
procedure PUT(x)
3   buf[nextprod] ← x
4   nextprod ← (nextprod + 1) mod cap
5   count ← count + 1
function GET()
6   x ← buf[nextcons]
7   nextcons ← (nextcons + 1) mod cap
8   count ← count − 1
9   return x

Public Methods:
procedure PRODUCE(x)
10  while true do
11      TXBEGIN()
12      if Full() then
13          notfull.CONDWAIT() //or RETRY()
        else
14          PUT(x)
15          notempty.CONDSIGNAL()
16          return
17      TXCOMMIT()

function CONSUME()
18  while true do
19      TXBEGIN()
20      if Empty() then
21          notempty.CONDWAIT() //or RETRY()
        else
22          item ← GET()
23          notfull.CONDSIGNAL()
24          return item
25      TXCOMMIT()

A Dangerous Scenario:
procedure Produce1Consume2(x)
26  TXBEGIN()
27  inprogress ← true
28  x ← CreateElement()
29  PRODUCE(x)
30  a ← CONSUME()
31  b ← CONSUME()
32  Use(a, b)
33  inprogress ← false
34  TXCOMMIT()
```

T will not wake until some subsequent call to PRODUCE occurs. During the interval until then, the temporary value of $inprogress$ will be visible. Furthermore, before T wakes, any number of other threads may produce and consume an unbounded number of elements, such that when T finally wakes and consumes another element, it will not be certain that it consumed two consecutively produced elements.

The mechanisms we propose avoid this problem. We replace lines 13 and 21 with calls to one of our mechanisms, and then T will be completely rolled back when it reaches CONSUME line 21. Atomically with its rollback, T will publish information that allows subsequent transactions to decide, after they commit, whether program state has changed in a way that justifies the re-execution of T.

Aesthetically, our mechanisms are cleaner than condition variables: there is no need to explicitly signal (lines 15 and 23), nor risk of forgetting to do so. Furthermore, the loops on lines 10 and 18 are unnecessary, since transaction abort provides an implicit back-edge.

Figure 1 illustrates the behavior of the mechanisms we discuss in this paper. At time 1, $Waiter$ reaches an untenable state (line 20), undoes its effects, and sleeps. At this time, the state of the programs' memory is indistinguishable from before $Waiter$ began, but $Waiter$ has published a representation of the precondition on which it depends. At time 2, some other producer ($Writer$) commits, and establishes that the precondition needed by $Waiter$ now holds. Therefore, at time 3, $Waiter$ wakes and then runs to completion (time 4). RETRY, AWAIT, and the WAITPRED conditional synchronization mechanism we introduce in this paper, achieve this behavior. In contrast, with monitors, at time 1, $Waiter$ would commit its changes to shared memory. At time 2, $Writer$ would signal a condition variable, and then at time 3, $Waiter$ would begin a new transaction by checking if the precondition holds, and if so, continuing to its commit point at time 4. The key differences are (a) whether atomicity is preserved, and (b) which thread is responsible for checking the precondition on which $Waiter$ depends.

3. The Original Retry Mechanism

Our goal is to provide a mechanism that (a) works with HTM, (b) supports RETRY-style condition synchronization, and (c) can be used to implement other conditional synchronization techniques. We first consider an implementation of RETRY that is faithful in spirit to prior STM proposals for Haskell [15] and C++ [31]. Algorithm 1 assumes an eager STM with in-place updates, such as TinySTM [11], or GCC-TM [12]. Addresses are mapped to entries in a table of locks, so that on every read by an in-flight transaction, the legality of the read can be determined by reading the lock, and saving its location for later validation. On every write, the corresponding lock is acquired, the old value stored in an undo log, and memory is updated directly.

Algorithm 1: Original Retry mechanism, adapted from [15] to use eager STM. For simplicity of presentation, a lock prevents concurrent accesses to the list of waiting threads.

```
   Per-Thread Metadata
     reads   : lock*              // locks for locations read by txn
     undos   : ⟨addr, val⟩*       // Writes by this transaction
     locks   : lock*              // locks for locations written by txn
     sem     : semaphore          // per-thread semaphore for RETRY
     cp      : Checkpoint         // Used on abort/rollback
   Global Metadata
     waiting : Thread*            // list of sleeping threads
   procedure RETRY
       // undo writes
 1     undos.undoAll()
       // release locks as if transaction never ran
 2     locks.resetAll()
       // atomically add calling transaction to waiting if still valid
 3     waiting.lock()
 4     if reads.valid() then
 5         waiting.insert(self)
 6         waiting.unlock()
 7         sem.wait()
 8     else
 9         waiting.unlock()
       // restart the transaction
10     reads ← undos ← locks ← {}
11     cp.restore()

   procedure TXCOMMIT
       // handle read-only transactions
 1     if readOnly() then
 2         reads ← {}
 3         return
       // fail if reads not valid
 4     if ¬reads.valid() then
 5         undos.undoAll()
 6         locks.releaseForAbort()
 7         reads ← undos ← locks ← {}
 8         cp.restore()
       // transaction is valid... release locks
 9     locks.releaseForCommit()
       // check for transactions to wake
10     waiting.lock()
11     for e ∈ waiting do
12         if e.reads ∩ locks then
13             waiting.remove(e)
14             e.sem.signal()
15     waiting.unlock()
       // reset lists
16     reads ← undos ← locks ← {}
```

The goal of RETRY is to undo the effects of a transaction and delay subsequent re-attempts until there is a chance that re-executing it would be profitable. Re-execution is delayed until some later transaction performs a write that overlaps with a read of the retrying transaction: since a re-execution would then observe different state, there is a chance that

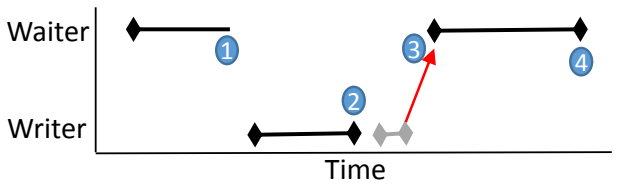

Figure 1: High-level interaction between a waiter and a writer. At time 2, the writer has changed the shared state in a way that makes re-attempting the waiter worthwhile. The decision to wake the waiter occurs at time 3.

re-execution is worthwhile. The challenge is to ensure that while a thread is being put to sleep, its reads are not modified in a manner that necessitates a wakeup. In STM implementations that provide opacity [13], the calling transaction is guaranteed that its reads are valid at the time when RETRY is called. The implementation must ensure the transaction's reads remain valid while adding itself to the list of waiting transactions. This necessitates some manner of validation atomic with the update to $waiting$ (lines 3–9 of RETRY). While accidental wakeups are harmless, there can be subtle races if two writers are simultaneously attempting to wake a transaction, and the transaction resumes and modifies $reads$ concurrently with a thread executing line 12 of TXCOMMIT. For clarity of presentation, Algorithm 1 employs a global lock. Our implementation achieves greater concurrency by using an ad-hoc nonblocking technique to protect accesses to $waiting$.

With regard to supporting RETRY in HTM, there is a second challenge. Past implementations of RETRY perform intersections over sets of locks, instead of sets of actual addresses read and written. Clearly, HTM does not have such locks, and Hybrid TM appears to have converged on designs without locks [6, 21, 27]. However, even if the implementation were to switch to using address/value pairs (which would also prevent silent stores from causing fruitless wakeups), the mechanism would remain incompatible with HTM. Even though the wakeup routine occurs after a writing transaction has logically committed, it requires access to the list of locations written by the committed transaction, and today's HTM systems do not provide this information.

4. A Low-Level, HTM-Friendly Mechanism

To construct an HTM-friendly mechanism that is compatible with HTM, it is instructive to focus on the high-level behavior of RETRY. In Figure 1, we see a rough sketch of the interaction between a waiting transaction that calls RETRY and a writing transaction that causes the waiter to wake.

Note that at time 1, the waiter does not commit changes to program state, but does update program metadata. In today's HTM systems, which lack support for escape actions [38], it appears impossible to achieve this behavior in HTM. However, since retrying is not on the critical path of the application, re-executing that transaction in a software mode with escape actions does not seem onerous. Additionally, the wakeup operation by the writing transaction, at time 3, occurs strictly after the transaction commits its changes to shared memory. Wakeup is not atomic with writer commit.

Algorithm 2: Abstract mechanism for descheduling transactions

```
Additional Per-Thread Metadata
  asleep      : Boolean    // True if thread has been woken
  waitfunc    : Function   // Decides if thread should wake
  waitparams  : Record     // Parameters to waitfunc

pre-condition : Function called from a transactional context
input         : A function (f) that returns a Boolean, and a
                record (p) encapsulating the parameters to f
procedure DESCHEDULE(f, p)
    // roll back the transaction
1   undos.undoAll()
2   locks.resetAll()
3   reads ← undos ← locks ← {}
    // preserve the checkpoint
4   tmpCp ← deepCopy(cp)
    // begin an outermost transaction
5   wait ← false
6   TxBegin()
7   if ¬f(p) then
        // precondition does not hold... go to sleep
8       waitfunc ← f
9       waitparams ← p
10      asleep ← true
11      waiters ← waiters ∪ self
12      wait ← true
13  TxCommit()
    // If f returned false, sleep
14  if wait then
15      sem.wait()
        // on wakeup, prevent future notifications
16      TxBegin()
17      waiters ← waiters − self
18      TxCommit()
    // restart the parent transaction
19  tmpCp.restore()

pre-condition : Function called during TXCOMMIT, after the
                transaction has committed
procedure wakeWaiters()
    // use a transaction to copy the set of waiting threads
1   TxBegin
2   l ← waiting.copy()
3   TxCommit
    // check each entry's condition
4   for e ∈ l do
5       shouldWake ← false
6       TxBegin
7       if e.asleep ∧ e.waitfunc(e) then
8           e.asleep ← false
9           shouldWake ← true
10      TxnCommit
11      if shouldWake then e.semaphore.signal()
```

Our technique treats the wakeup operation as a computation over shared memory. Algorithm 5 provides more detail. We say that a thread wishing to delay its execution will call DESCHEDULE. DESCHEDULE undoes the effects of the transaction. It then uses a transaction to evaluate some read-only transactional function f using parameters p. If the function returns $true$, then the calling transaction is immediately restarted. Otherwise, the thread makes f and p visible to other threads and then puts itself to sleep. By expressing the rescheduling condition as $f(p)$, we need not validate before adding the caller to the list of waiting transactions. Instead, we can place the caller in the list, and then double-check the condition. This would not be possible if we were relying on the underlying TM's metadata.

After any writing transaction commits, it calls $wakeWriters$ to find any sleeping threads whose transactions could now complete. If the computation to wake a thread is not too complex, then lines 6–10 should be able to execute as a hardware transaction. Consequently, we must avoid contention (e.g., by making a shallow copy of the list of waiting threads) and eschew escape actions (e.g., by deferring semaphore operations until line 11). A minor implementation detail to note is that, as of line 3 of DESCHEDULE, the calling transaction is completely undone. While we can safely call new transactions, we must ultimately restart the calling transaction, and thus it is necessary to preserve the calling transaction's checkpoint (line 4 and line 19).

5. Implementing Condition Synchronization with Deschedule

5.1 Implementing Retry

As discussed in Section 4, HTM without escape actions appears unable to atomically undo its effects and publish itself into a list of waiting threads. Thus in our RETRY implementation, a hardware transaction that encounters the RETRY keyword will restart in software mode.[1] In the software mode, on every read, the address and value produced by the read are logged to a special $waitset$. These behaviors are shown in $TxRead$, Algorithm 3. Since the retrying transaction would otherwise immediately make a system call to put its calling thread to sleep, we see this switch-and-restart behavior as a form of backoff. In the best case, the transaction will discover, on re-execution, that its precondition has been established by a concurrent writing transaction.

If the thread is in software mode, the next challenge is to ensure that it can announce an operation that can be checked by hardware transactions. Regardless of the metadata in the underlying TM system, we always use values to implement RETRY. If necessary, we restart the transaction to ensure that it logs values on every $TxRead$, so that it can express the precise state it observed when it next reaches RETRY.

[1] Existing best-effort HTM implementations already require this fallback path to overcome transactional capacity limitations [35].

Algorithm 3: An implementation of Retry based on Deschedule

```
   Additional Per-Thread Metadata
     is_retry     : Boolean     // True if thread called retry
   pre-condition     : Function called from within a transaction
   procedure findChanges(Tx)
1      for ⟨a, v⟩ ∈ Tx.waitset do
2          if *a ≠ v then return true
3      return false
   procedure TxRead(addr)
1      if is_retry then
2          if addr ∈ undos then
3              v ← undos.get(addr)
4          else
5              v ← *addr
6          waitset.append(⟨addr, v⟩)
       ...// original TxRead code follows
   procedure RETRY()
       // ensure software mode
1      if HTM_mode() then restart_in_STM()
       // if waitset not populated, restart and populate it
2      if ¬is_retry then
3          is_retry ← true
4          waitset ← reads ← undos ← locks ← {}
5          cp.restore()
       // use DESCHEDULE to suspend transaction
       else
6          is_retry ← false
7          DESCHEDULE(findChanges, self)
```

In this manner, $findChanges(waitset)$ can precisely track whether the transaction should be resumed. Note, too, that a silent store (one that does not change the location's value) will not wake a thread.

If the transaction has populated $waitset$, then RETRY reduces to a call to DESCHEDULE($findChanges, self$). That is, RETRY will undo the transaction's effects, add the transaction to $waiting$, double-check that the values read by the failed attempt have not changed, and then put the thread to sleep. For simpler presentation, we omit code for cleaning up the is_retry flag, and we lazily reset the $waitset$.

5.2 Implementing Await

With RETRY, the programmer cannot restrict the set of addresses for which modifications will cause the transaction to re-execute. On the one hand, this is beneficial for deeply nested transactions, where it may be difficult to determine the precise locations that precede a wakeup. On the other hand, especially for shallow nesting, it may be desirable to limit the address range. In the Atomos language, Carlstrom et al. proposed the AWAIT keyword [4], which can be thought of as RETRY restricted to a single location. This restriction enables implementation inside of HTM, since the amount of data to track can be constrained. With RETRY al-

Algorithm 4: Algorithm for generalized Await

```
   pre-condition : Function called from a transactional context
   input         : A set of addresses
   procedure AWAIT(addrs)
       // roll back writes, so we can see original state of memory
1      undos.undoAll()
       // populate waitset
2      waitset ← undos ← {}
3      for a ∈ addrs do
4          waitset.append(⟨a, TxRead(a)⟩)
       // use DESCHEDULE to suspend transaction
5      DESCHEDULE(findChanges, self)
```

Algorithm 5: Algorithm for WaitPred

```
   pre-condition : Function called from a transactional context
   input         : A function to evaluate, and a set of parameters
   procedure WAITPRED(pred, args)
       // populate waitset
1      for arg ∈ args do
2          waitset ← arg
       // use DESCHEDULE to suspend transaction
3      DESCHEDULE(pred, self)
```

ready available, it is relatively straightforward to also implement this limited interface, as depicted in Algorithm 4.

Our implementation supports waiting on changes to an arbitrary number of memory locations, as indicated by the parameter $addrs$. In contrast to RETRY, where the addresses of the read set may not be known at the time of the call to RETRY (e.g., if the underlying TM implementation logs lock locations), with AWAIT, the programmer provides a list of addresses. Thus as long as we can see the contents of memory from the time when the transaction began, we can construct the correct initial values for AWAIT without restarting the transaction. There is a subtlety, however: while we must not read the speculative writes of the current transaction (and hence must undo writes first), we must also be sure that reads of those addresses are consistent with the entire transaction.

Our implementation assumes that the addresses passed to AWAIT had been previously read by the transaction, and the TM is opaque. In this case, we can re-use the existing code for reading memory within transactions ($TxRead$, line 3) to populate waitset: if that read returns a different value than the prior read to the same location, then the transaction will abort. Note that holding locks while performing the re-reads is necessary, due to the way that production STM is implemented: a read followed by a write may be executed as a "read for write" [22], in which case the address is not logged in the transaction's read set, only its write set. For such reads, and for certain TM implementations (such as timestamp extension [25]), releasing locks would be incorrect.

Note that it is possible that addresses passed to AWAIT were allocated by the transaction, as "Captured Memory" [9, 26]. Thus we must be careful about how we roll back the transaction. If it had allocations, those allocations cannot be undone until after the awaiting thread has been awoken by a subsequent writer. This concern also applies to RETRY.

5.3 Synchronizing with Explicit Predicates

Having developed support for RETRY and AWAIT through the use of DESCHEDULE, we now introduce WAITPRED. The idea behind WAITPRED is to replace $findChanges$ with user-specified functions. In this manner, it is possible to avoid wakeups that occur when an address is written, but the written value is not one that establishes the needed precondition for the waiting transaction. The only challenge with our implementation is that the user-specified predicate function may expect arguments to be passed to it (e.g., the address of a specific bounded buffer). We cannot construct an object to store these arguments, since the writes might be undone during DESCHEDULE. Instead, we receive a variable number of arguments, which the library then marshals into the $waitset$. Details appear in Algorithm 5.

5.4 Discussion

There several high-level aspects of our design that merit additional discussion. First, we note that HTM is immediately usable for non-rescheduling transactions, since their only change is to execute $wakeWaiters$. For hardware transactions that must be descheduled, the lack of escape actions requires that we change modes and then re-execute the transaction in software. For WAITPRED, in limited cases re-execution is not needed. For example, Intel's hardware transactional memory support allows the programmer to emit an 8-bit value to describe any explicit self-abort. If the total set of reschedule function/parameter combinations is less than 255, then it is possible to use this value as an index into a table, so that the hardware transaction can abort, enqueue its predicate, and then put itself to sleep.

Second, we note that our algorithms appear to be compatible with all known abortable single-version STM algorithms. Support for lazy TM, TM with varying metadata implementations, and TM with visible reads are all straightforward modifications to the algorithms presented herein. HyTM algorithms are similarly straightforward to support. Furthermore, our algorithms are general enough to handle production-level TM systems, such as GCC, which use read-for-write and other optimizations.

Third, our mechanisms do not require garbage collection, and ensure that explicit allocation and reclamation remain safe. We are also careful to avoid erroneous wakeups. For example, in AWAIT, we do not store values into the $waitset$ during $TxRead$, instead ascertaining these values after the undo log has been rolled back. To do otherwise could result read-after-write operations populating the log with values that were essentially produced out of thin air. Every subsequent writer commit might then wake the transaction.

There is one caveat: while the mechanisms cleanly express conditional synchronization as predicates over states, there can be unexpected scheduling outcomes. Suppose transaction T_A reschedules to await a list becoming nonempty. Let transactions T_B, T_C, and T_D insert, remove, and insert elements into the list, respectively, with each completing a commit but stalling before calling $wakeWaiters$. In this case, any might be the one to successfully wake T_A, and there is no relationship between the identity of the transaction that established the condition upon which T_A waited, and the identity of the transaction that awoke T_A. Similarly, suppose the absence of T_D: in this case, if T_B completes $wakeWaiters$ before T_C commits, then it is possible for T_A to wake, T_C to commit, and T_A to ultimately observe an empty list and go to sleep again. These examples illustrate the importance of becoming comfortable thinking of conditional synchronization as predicates over program states.

6. Programmability

Programming with WAITPRED and AWAIT is similar to programming with RETRY. Within a transaction, programmers test if a necessary condition does not hold, and use the appropriate command to unroll any pending writes by their transaction and put the calling thread to sleep. Figure 2 shows the changes to Listing 1 necessary to use the any of the three mechanisms we discuss. The programmer can choose to wait on a given condition specified by a function (left column), a static address list (middle column), or the dynamic set of addresses read by the transaction (right column). In all three cases, there is a slight decrease in code and control flow, relative to condition variables.

The first question we consider is whether this plurality of mechanisms will be valuable to programmers. There is a difference in the complexity of reasoning required when using WAITPRED, AWAIT, and RETRY. Consider the $Produce1Consume2$ example from Section 2. In this case, the use of RETRY within the Put and Get methods is sufficient, but the use of WAITPRED is not: the designer of the bounded buffer would likely use the condition $\neg Empty()$ as the predicate in $Consume$, when atomic consumption of two elements would require the predicate $count \leq (cap-2)$. Similarly, if the code were atomically consuming a total of two elements, from up to two buffers, then we might need to AWAIT using state encapsulated in two different objects. Roughly speaking, there appears to be a tradeoff between the generality of the conditional synchronization mechanism, and run-time overhead: WAITPRED should avoid unnecessary wakeups; AWAIT avoids validation of a full read set; and RETRY provides generality in the face of composition.

The second question is whether composable conditional synchronization is valuable. If we consider only existing code, the answer is clearly negative. For example, we analyzed 16 open-source applications and benchmarks that use condition variables, and found that in every single case, no more than one lock was held at the time that `condvar.wait()` was called. Furthermore, in every case, the wait was at the same lexical scope as lock acquisition and release: waiting was never even done in a function called from the critical section. We suspect this to be more a consequence of the difficulty of using condition variables than the lack of a need for composable condition synchronization; while solutions to the nested monitor problem are well known [33], they have not been adopted. Instead, programmers are taught early in their careers to avoid blocking in nested monitor calls, and they appear to obey this lesson.

The third question we ask is why preserving atomicity is useful. Clearly, there are forms of inter-thread condition synchronization that require breaking atomicity. For example, the classic two-wait reusable barrier [2, Chapter 5] cannot be implemented easily with transactions. This is a known problem [30], but perhaps overstated: for example, we have not observed any code that uses a barrier within a critical section. On the other hand, there are many uses of condition variables that do not require breaking atomicity. For example, in PARSEC, converting from condition-variables to deschedule-based mechanisms was straightforward. In many cases, we could replace each condition variable with an integer. To wait, we could then use AWAIT, passing the value of the integer, and to signal, we needed only to increment the integer.[2] In a few cases, a small change to control flow was also necessary, but as Table 1 shows, the overall impact on the code base was minimal.

In the end, we believe that the real value of composition and atomicity is yet to be discovered, partly because traditional mechanisms have made these features difficult (if not impossible) to use correctly. There is an existential argument that the abundant use of RETRY in Haskell [34] shows the potential of these mechanisms. In addition, new techniques are emerging, e.g., to simplify I/O from transactions [37], whose correctness depends on a condition synchronization mechanism that is composable and atomicity-preserving.

7. Evaluation

In this section, we evaluate the performance of our mechanisms on STM and HTM workloads. We are interested in two questions:

- How do our implementations compare to the current state of the art (i.e., transactional condition variables)?
- Do WAITPRED and AWAIT offer a performance benefit over RETRY?

To explore these questions, we run two categories of experiments. First, we use a bounded buffer micro-benchmark, which we parameterize based on the buffer size, number of producers, and number of consumers. This allows us to evaluate overheads in a situation where the condition synchronization mechanism is potentially used with high frequency,

[2] Strictly speaking, this is a broadcast, not a signal, but it does not compromise correctness.

```
1  procedure PutPred(x)
2      TxBegin()
3      if Full() then
4          WaitPred(¬Full, void)
5      Put(x)
6      TxCommit()

1  procedure PutAwait(x)
2      TxBegin()
3      if Full() then
4          Await(⟨&count⟩)
5      Put(x)
6      TxCommit()

1  procedure PutRetry(x)
2      TxBegin()
3      if Full() then
4          Retry()
5      Put(x)
6      TxCommit()

1  procedure GetPred()
2      TxBegin()
3      if Empty() then
4          WaitPred(¬Empty, void)
5      return Get()
6      TxCommit()

1  procedure GetAwait()
2      TxBegin()
3      if Empty() then
4          Await(⟨&count⟩)
5      return Get()
6      TxCommit()

1  procedure GetRetry()
2      TxBegin()
3      if Empty() then
4          Retry
5      return Get()
6      TxCommit()
```

Figure 2: Put and Get methods for a bounded buffer using three transactional condition synchronization mechanisms.

and where there may be more threads than cores. Second, we measure performance on the PARSEC benchmark suite [3]. We limit our evaluation to the 8 PARSEC benchmarks that use condition variables.

We compare 7 condition synchronization mechanisms: the baseline system, **Pthreads**, uses pthread locks to protect critical sections, and pthread condition variables for condition synchronization. **TMCondVar** is a transliteration of **lock**: transactions protect critical sections, and transaction-safe condition variables [32] provide condition synchronization. Note that these two implementations both break atomicity when a critical section waits. **WaitPred**, **Await**, and **Retry** correspond to our mechanisms, built upon a single HTM-friendly mechanism. **Retry-Orig**, used only in STM experiments, is a good-faith implementation of RETRY from [15]. Finally, **Restart** aborts and immediately restarts a transaction any time a precondition does not hold.

Our experiments consider three configurations: Eager STM corresponds to a configuration in which transactions are provided via STM, using the default GCC "ml-wt" implementation (a privatization-safe variant of TinySTM with undo logs [11]). Lazy STM is like Eager STM, but uses redo logs [8]. HTM corresponds to a configuration in which transactions are provided via HTM, using the GCC "htm" implementation. Our experimental system had a single Intel Core i7-4770 CPU running at 3.40GHz. The i7-4770 has four cores, each 2-way multi-threaded, for a total of 8 hardware threads, and supports hardware TM through Transactional Synchronization Extensions (TSX). The software stack included Ubuntu 14.04, Linux kernel 3.13.0-43, and GCC 5.0.0, with -O3 optimizations.

7.1 Producer Consumer Micro-benchmark

We begin with bounded buffer micro-benchmark experiments, based on the Figure 2. There are three configuration parameters: the size of the buffer, the number of producers, and the number of consumers. Each benchmark trial entails 2^{20} total elements produced, with an equal number of operations assigned to each producer, and 2^{20} elements consumed, with an equal number of operations assigned to each consumer. We half-fill the buffer before each experiment.

Figures 3–5 present the results for STM and HTM executions. In each chart, pi-cj refers to an execution with i producers and j consumers. The X axis reflects the size of the buffer (4, 16, or 128 elements). Values are the average of 5 trials. Variance was generally low, though there are some exceptions, discussed below.

Our first observation is that, for these microbenchmarks, simply retrying the transaction immediately offers the best performance. This is a consequence of the simplicity of the microbenchmark, and does not apply to PARSEC. We do not discuss **Restart** further in this subsection.

When producers and consumers are balanced and there is no oversubscription (p1c1, p4c4), our mechanisms outperform condition variables: they avoid calls into the pthread library, and wakeups only affect a small number of threads. The effect is most pronounced for small buffers, where sleeping is more likely. Furthermore, our mechanisms tend to outperform the original retry technique, suggesting that separation of the mechanism from the underlying TM implementation does not introduce significant overhead.

This same relationship mostly holds for small amounts of imbalance (p1c2, p2c4). However, in these cases there is a higher incidence of sleeping and waking up. Consequently a few new behaviors emerge. First, STM and HTM behave differently. This is a consequence of the TM implementations: In HTM, conflicts between the read-only $wakeWaiters$ call and the execution of other transactions can result in aborts. This is due to TSX aborting transactions on some read-write conflicts that do not cause aborts in the STM implementations. Second, and more significantly, our mechanisms have a higher frequency of wakeups. While the pthread baseline will only wake one thread after any production or consumption, our mechanisms essentially broadcast. This can result

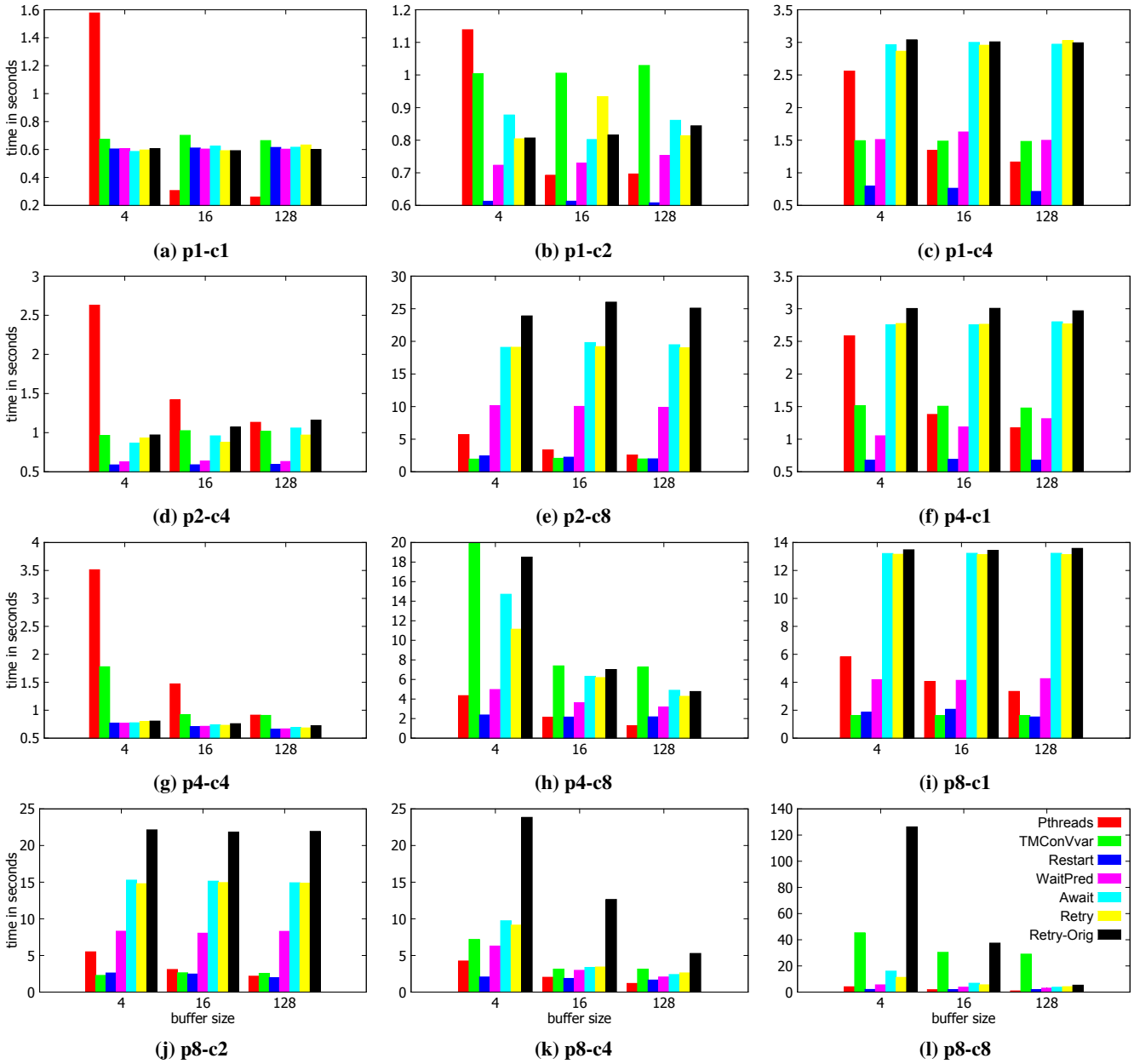

Figure 3: Bounded buffer performance with eager STM.

in pathological behaviors for homogeneous and imbalanced micro-benchmark configurations: consider a production in p1c4 when the buffer is empty. After the production, 4 consumers are woken. They all contend for the same element, one succeeds, three fail, and then the failed threads go back to sleep. The effect is equivalent to using `broadcast` with pthread condition variables, and is inherent to our three techniques. Third, we observe that the latency differences between our mechanisms meet our expectations: WAITPRED is less costly than AWAIT, but since transactions are tiny, AWAIT does not offer an advantage over RETRY: only one fewer location is tracked.

Under imbalance (p4c1, p1c4, p8c1, p8c2, p2c8), these trends continue. However, particularly when there is oversubscription (either 8 producers or 8 consumers), an additional trend arises: transactional condition variables show a significant advantage. There is a synergy, in that transactional condition variables both (a) allow concurrency among the critical sections of producers and consumers (locks do not), and (b) do not perform broadcast wakeups, which would cause context switching. Note, however, that these results begin to have higher variance (between 0.1 and 1), due to the impact of preemption and context switches when there are more threads than cores.

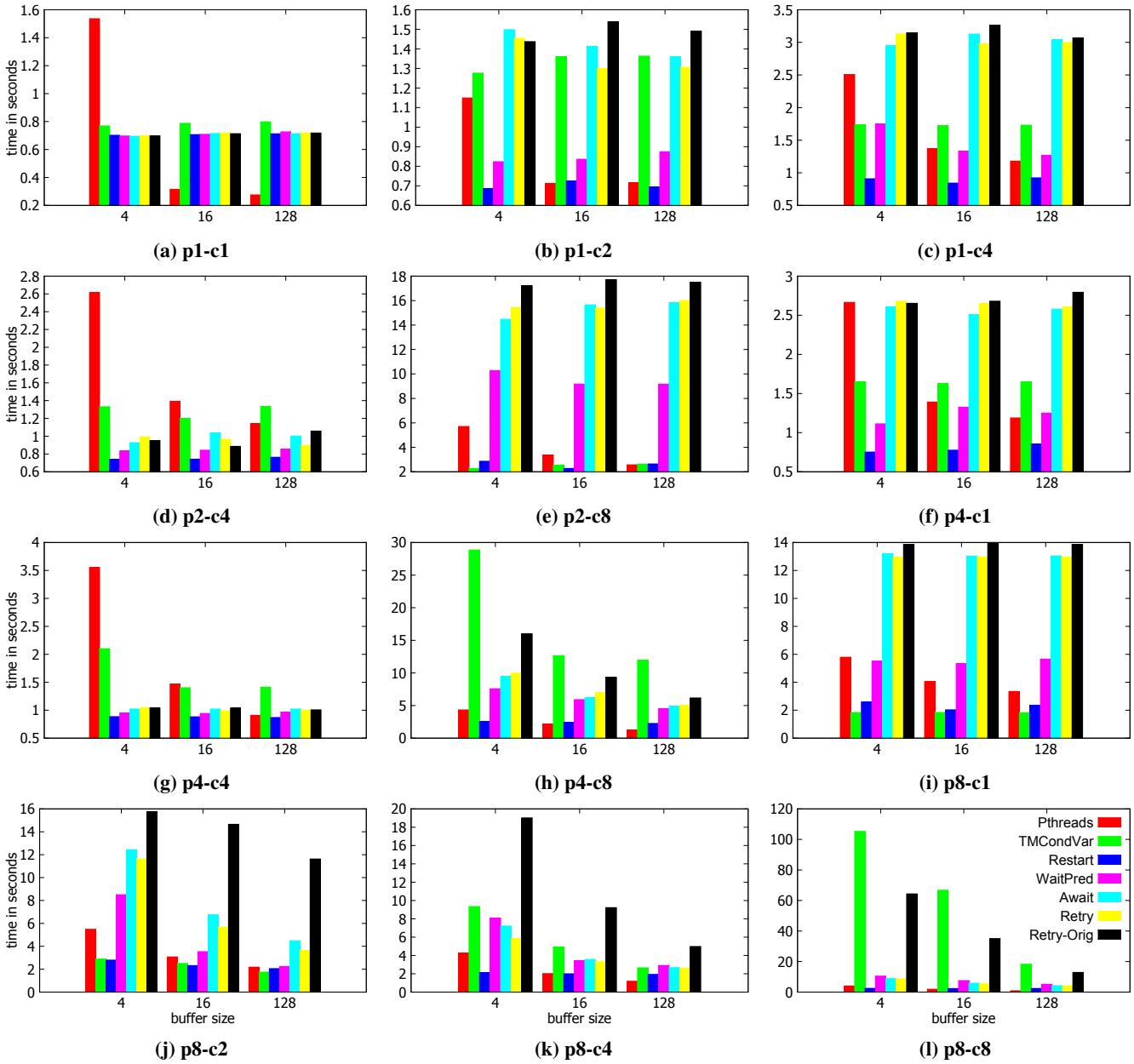

Figure 4: Bounded buffer performance with lazy STM.

For HTM, oversubscription experiments (p8c4, p4c8, p8c8) behave the same as under imbalance. HTM implementation details matter here: To ensure progress, the GCC HTM implementation suspends concurrency after a transaction aborts twice, so that it may execute to completion. Additionally, when a hardware transaction calls our mechanisms, we suspend concurrency so that the transaction can run in a software mode that allows for escape actions. In STM there is one difference: transactional condition variables experience pathologically bad behavior. When too many producers or consumers run simultaneously, the benchmark can wind up in a situation where all but one thread is asleep; at this point, the roughly $3\times$ latency overhead of STM instrumentation dominates, and variance also increases to above 1.

7.2 PARSEC Performance

We now turn our attention to the performance of our mechanisms on larger applications. We consider the eight PARSEC benchmarks that make use of condition synchronization. Table 1 describes the number of lines of code that were removed from each benchmark to eliminate condition variables, and the number of lines that were added to synchronize threads via our mechanisms.

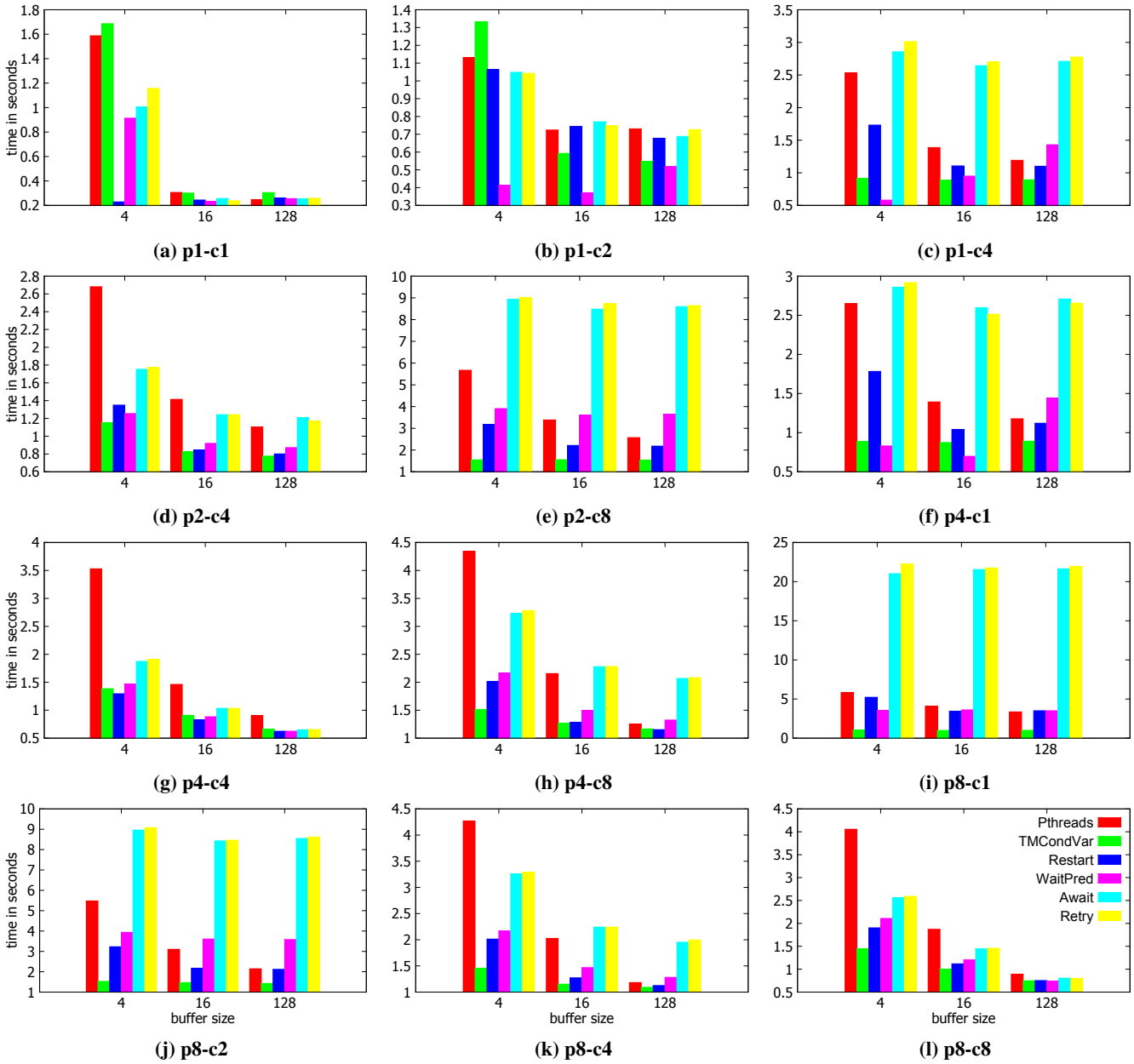

Figure 5: Bounded buffer performance with HTM.

The number of lines needed to use our mechanisms is comparable to the number of lines related to condition variables. Quantitatively, the changes are small and localized. Qualitatively, while the code using our mechanisms was typically as long as the condition variable code, it was usually simpler, since it did not have to worry about breaking the atomicity of transactions. Perhaps most surprisingly, WAITPRED did not require more code than our other mechanisms: the predicate functions we needed to write were either tiny, or already present in the program.

Figures 6–8 present PARSEC performance for STM and HTM. Each bar is the average of five trials; variance was uniformly low. Note that some benchmarks only execute for thread counts that are even or powers of two. We used the transactional version of PARSEC provided by Wang et al. [32]. This has two immediate consequences: first, we observe a slight slowdown for transactions (TM) versus pthreads (lock). This is not surprising: PARSEC is carefully tuned lock-based code, and the TM version simply replaces locks with critical sections, without any careful tuning. Second, dedup performs very poorly with TM. This, too, is expected: dedup performs I/O within critical sections; the TM runtime forbids concurrency during transactions that

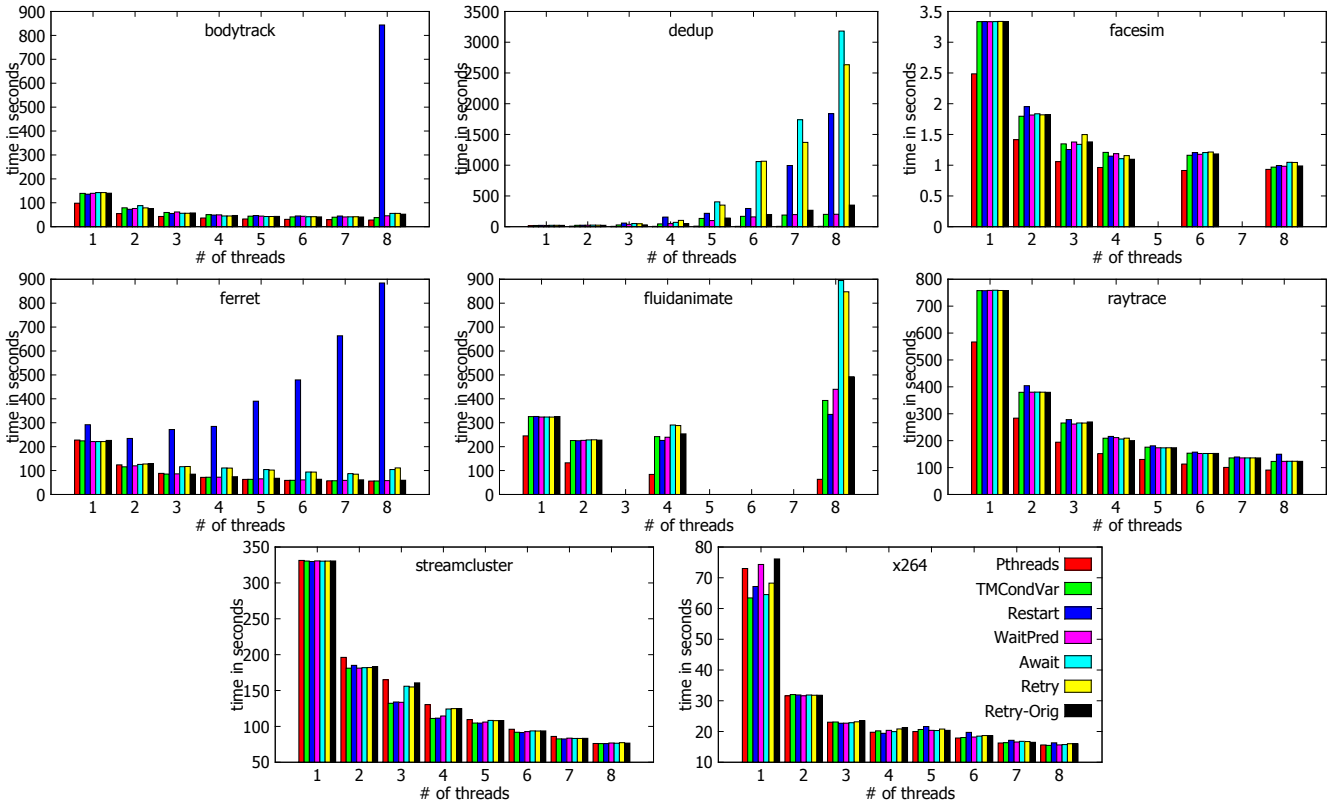

Figure 6: PARSEC performance with eager STM.

Benchmark	WAITPRED	AWAIT	RETRY	Removed
bodytrack (5)	47	55	47	54
dedup (3)	66	88	66	71
facesim (7)	47	55	47	38
ferret (2)	31	49	31	47
fluidanimate (4)	60	68	60	126
raytrace (3)	76	88	76	38
streamcluster (5)	70	82	70	139
x264 (1)	15	21	15	14

Table 1: Lines of code added and removed for different condition synchronization mechanisms in PARSEC. Numbers in parentheses represent unique condition synchronization points for each benchmark.

perform I/O, to avoid conflicts/rollback after I/O has been performed but before the transaction has committed.

A few broad trends that emerge from these experiments. First, the performance difference between transactions with condition variables and our mechanisms is negligible: in real-world programs, where condition synchronization overheads do not dominate, the cost of synchronizing is not significant. Second, the performance of AWAIT tends to mirror the performance of RETRY. This outcome is due to the larger sizes of our transactions: AWAIT effectively prunes the set of locations on which a sleeping transaction waits. This, in turn, reduces overhead in $wakeWaiters$, saving time after every transaction commit that overlaps with a sleeping thread. Lastly, we observe subtle variations in the relative merit of different synchronization mechanisms for different thread counts. For example, fluidanimate's performance at 8 threads depends more substantially on the condition synchronization mechanism than it does at 2 threads.

Overall, the outcome is that performance with our mechanisms is acceptable, and thus the main question is whether it is more appealing to programmers. For the most part, we believe the answer is affirmative. Especially in codes like PARSEC, where macros and compile-time options obfuscate control flow, there is a significant advantage to a condition synchronization technique that does not break atomicity; it is already difficult to reason about critical sections in PARSEC. This is even more true for library code, which may involve nested transactions and condition synchronization.

Unfortunately, our mechanisms do not obviate condition variables: It is not correct to explicitly abort a transaction after it has performed I/O. In the C++ Draft TM Specification [1], such transactions are distinguished, lexically, as "relaxed transactions". A strict approach would forbid our techniques in relaxed transactions. In cases like dedup, where condition synchronization occurs before I/O, our implementations remain correct. However, if a critical section must perform condition synchronization after I/O, then it cannot use our mechanisms, and must use condition variables.

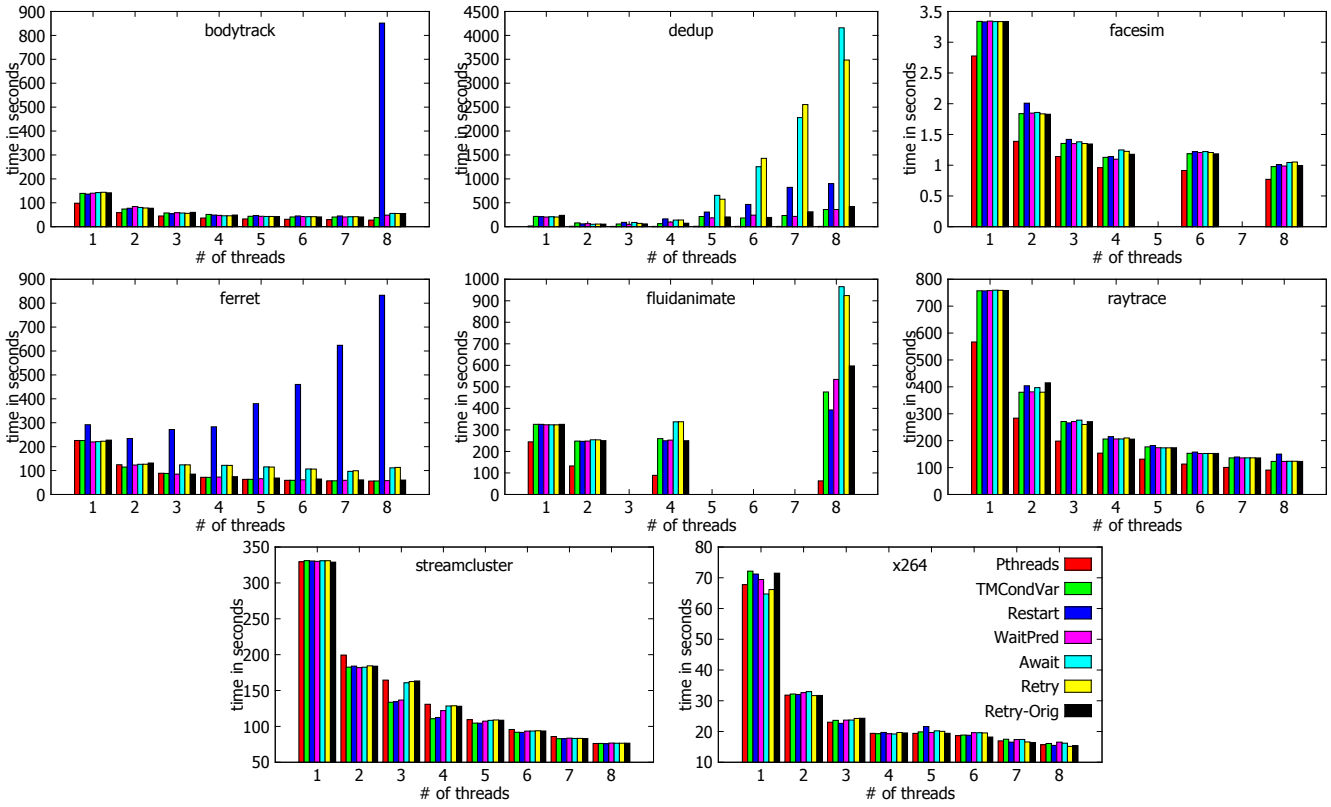

Figure 7: PARSEC performance with lazy STM.

8. Related Work

There are a few strategies for managing condition synchronization in TM, which we outline below. The most obvious is to make condition variables compatible with transactions. This approach has the benefit of simplifying the transactionalization of legacy code, and was identified as an important challenge by Ringenburg and Grossman [28] and Yoo et al. [36]. Dudnick and Swift subsequently presented hardware and OS extensions that allowed for transaction-safe condition variables [10], and Yoo et al. later showed that existing hardware could be made compatible with condition variables through a novel use of Linux futexes [35]. Wang et al. subsequently proposed an OS-agnostic, hardware-agnostic mechanism for transaction-safe condition variables, but it required extensions to the compiler [32]. While our mechanisms employ a different programming model, and are thus somewhat incomparable, we note that these techniques avoid OS, hardware, and compiler modifications.

Harris and Fraser [14], and later the X10 group [5], suggested a Conditional Critical Regions style of synchronization, in which 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. Harris et al. later extended this to the RETRY mechanism [15], which we study in this paper. Our work separates the retrying mechanism from the underlying TM implementation, and lets the programmer express the precondition explicitly.

There are many other proposals for synchronizing transactions, though none have gained traction outside of the research community. Smaragdakis et al. proposed "punctuated transactions" as a means of handling I/O and condition synchronization [30]. Like condition variables, this approach breaks the atomicity of transactions; like our work, it also allows the programmer to specify precise predicates, which govern when the awoken transaction can resume and how it can re-establish atomicity. Proposals for synchronizing transactions via group commit were proposed by Luchangco and Marathe [20] and Lesani and Palsberg [17]. Luchangco later showed that this technique can approximate condition variables [19]. However, the techniques have high complexity and are not compatible with HTM [18].

9. Conclusions and Future Work

We introduced a new low-level technique that allows condition synchronization among transactions without sacrificing atomicity. By expressing wakeup conditions as transactional computations, we are able to support a wide variety of TM implementation techniques, to include hardware TM, without introducing overhead in the common case. Furthermore,

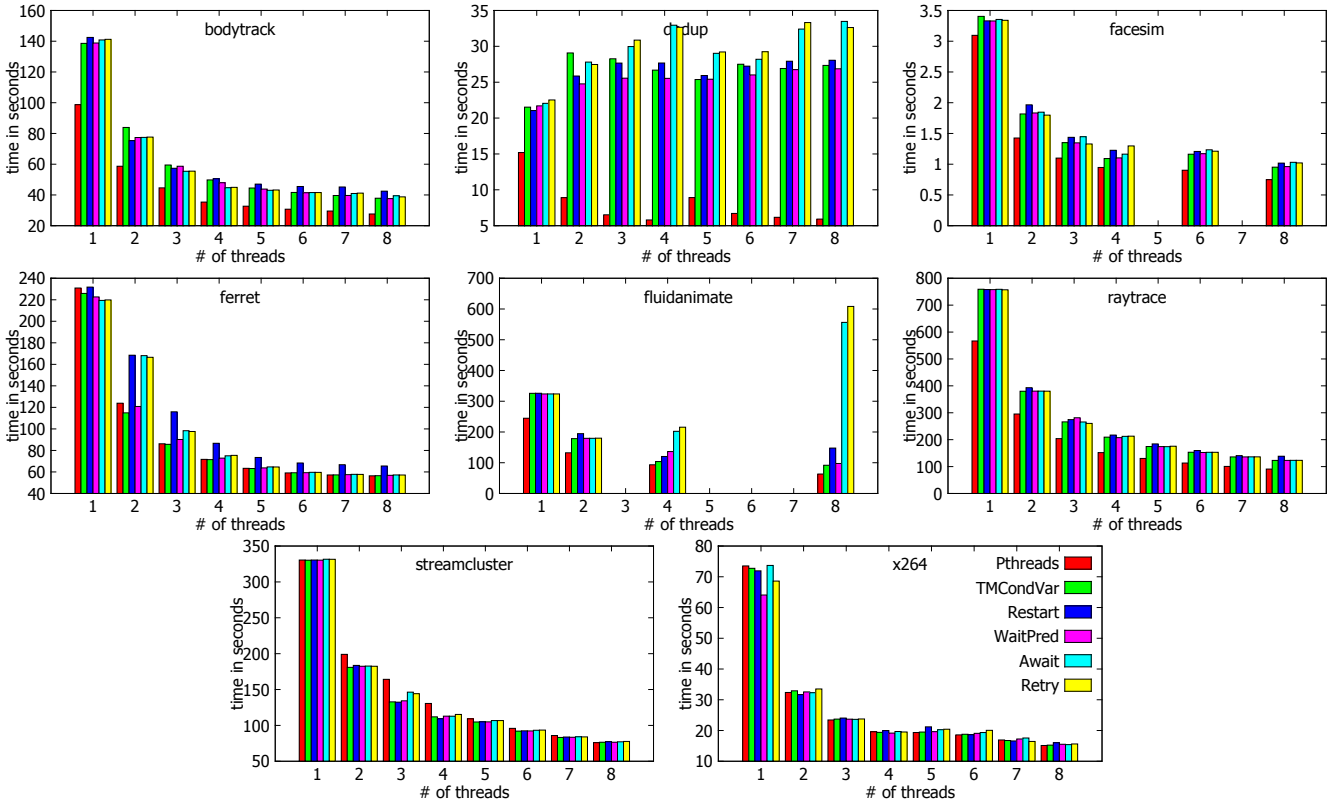

Figure 8: PARSEC performance with HTM.

by maintaining atomicity, we are able to support the composition of synchronizing transactions inside of other transactions. At the present time, we are not aware of programs that can take advantage of composition, but perhaps only because correct composition has been too difficult when using locks and condition variables. Our work will make it possible to explore the potential of composition to simplify the creation of complex concurrent programs. We expect such an exploration to be able to feed back into programming methodologies for lock-based code, and inform new techniques that could be compatible with nested monitors.

Our low-level technique can be used to construct several atomicity-preserving linguistic constructs. We provide implementations of RETRY [15] and AWAIT [4], as well as a new mechanism, WAITPRED, which enables wakeup to be contingent on an explicit programmer-provided predicate. These constructs differ in terms of performance and generality, with WAITPRED most likely to avoid spurious wakeups, but also hardest to use correctly when condition synchronization occurs in a nested transaction.

Our evaluation revealed negligible performance impact on large, well-tuned benchmarks like PARSEC. At the same time, the source code changes to use our mechanisms in place of traditional condition variables were minimal and straightforward. On stress-test micro-benchmarks, behavior was more nuanced, and in the end it seems that workload, machine, and TM implementation characteristics will affect which mechanism offers peak performance. Perhaps the most important observation from our evaluation is simply that performant condition synchronization is possible without breaking atomicity.

The most significant open issue with our work is studying the relationship between atomicity-preserving condition synchronization techniques and the "relaxed transactions" of the Draft C++ TM Specification. In our work, we observed a compatibility with relaxed transactions that have not yet performed I/O. Whether such compatibility can be statically enforced is an open question, as is the question of whether such compatibility is sufficient. Our experience with PARSEC was positive, but further workload and application usage studies will be necessary.

Acknowledgments

We thank our shepherd, Michael Swift, whose guidance greatly improved the quality of this paper. This material is based upon work supported by the National Science Foundation under Grants CAREER-1253362 and CCF-1218530. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

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] T. Anderson and M. Dahlin. *Operating Systems Principles & Practice*. Recursive Books, 2014.

[3] 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*, Toronto, ON, Canada, Oct. 2008.

[4] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*, 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 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] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. Scott, and M. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, Mar. 2011.

[7] L. Dalessandro, M. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[9] A. Dragojevic, Y. Ni, and A.-R. Adl-Tabatabai. Optimizing Transactions for Captured Memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[10] 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.

[11] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[12] Free Software Foundation. Transactional Memory in GCC, 2012. http://gcc.gnu.org/wiki/TransactionalMemory.

[13] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[14] 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.

[15] 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.

[16] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] A. Matveev and N. Shavit. Reduced Hardware NORec: A Safe and Scalable Hybrid Transactional Memory. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey, Mar. 2015.

[22] 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 *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, Oct. 2008.

[23] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, Sept. 2007.

[24] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th IEEE/ACM International Symposium on Microarchitecture*, Austin, TX, Dec. 2001.

[25] T. Riegel, C. Fetzer, and P. Felber. Time-Based Transactional Memory with Scalable Time Bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, California, June 2007.

[26] T. Riegel, C. Fetzer, and P. Felber. Automatic Data Partitioning in Software Transactional Memories. In *Proceedings of*

the 20th ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 2008.

[27] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, June 2011.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] C. Wang, Y. Liu, and M. Spear. Transaction-Friendly Condition Variables. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, Prague, Czech Republic, June 2014.

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

[34] R. Yates and M. Scott. A Hybrid TM for Haskell. In *Proceedings of the 9th ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Mar. 2014.

[35] R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High Performance Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, Nov. 2013.

[36] R. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[37] T. Zhou and M. Spear. The Mimir Approach to Transactional Output. In *Proceedings of the 11th ACM SIGPLAN Workshop on Transactional Computing*, Barcelona, Spain, Mar. 2016.

[38] C. Zilles and L. Baugh. Extending Hardware Transactional Memory to Support Non-Busy Waiting and Non-Transactional Actions. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, ON, Canada, June 2006.