View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Lehigh University: Lehigh Preserve

Lehigh University
Lehigh Preserve

Theses and Dissertations

2016

Practical Condition Synchronization for Transactional Memory

Chao Wang
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
Wang, Chao, "Practical Condition Synchronization for Transactional Memory" (2016). *Theses and Dissertations*. 2867.
http://preserve.lehigh.edu/etd/2867

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

Practical Condition Synchronization for Transactional Memory

by

Chao Wang

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science & Engineering

Lehigh University

Jan. 2016

Copyright

Chao Wang

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

Date

Thesis Advisor

Co-Advisor(if any)

Chairperson of Department

Acknowledgements

I would like to extend my sincere gratitude to my advisor, Michael Spear. He always lend instructive advice to me when I met with difficulties in my experiments. He also devoted extraordinary time and efforts to revise this thesis. I am deeply grateful for what he has done for the completion of this work.

Special thanks to Yujie Liu and Wenjia Ruan for their efforts in assisting me when I come across obstacles during my study in the concurrent programming area.

I am so deeply debted to my family - My mom, my dad, and my sister. Without your strong support accross the Pacific Ocean, this thesis would not be possible.

Contents

Acknowledgement **iv**

List of Figures **vii**

Abstract **1**

1 Introduction **2**

1.1 Transactional Memory 2

1.2 Condition Synchronization 3

2 Condition Synchronization Mechanisms **5**

2.1 RETRY 5

2.2 Deschedule - An abstract mechanism 7

2.2.1 A Motivating Example 7

2.2.2 An HTM-friendly mechanism 9

2.2.3 Implementing RETRY 11

2.2.4 Implementing AWAIT 12

2.2.5 WaitPred: Synchronization with Explicit Predicates 13

2.2.6 Discussion 14

2.3 Programmability 16
2.4 Evaluation 18
2.4.1 Producer Consumer Micro-benchmark 21
2.4.2 PARSEC Performance 24
2.5 Related Work 28

3 Conclusion and Future Work **36**

Bibliography **38**

Appendix A Software TM Implementation Details **45**

Vita **52**

List of Figures

2.1 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. 9
2.2 Put and Get methods for a bounded buffer using three transactional condition synchronization mechanisms. 16
2.3 Bounded buffer performance with eager STM. 19
2.4 Bounded buffer performance with lazy STM. 20
2.5 Bounded buffer performance with HTM. 24
2.6 PARSEC performance with eager STM. 25
2.7 PARSEC performance with lazy STM. 25
2.8 PARSEC performance with HTM. 26

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 thesis, 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.

Chapter 1

Introduction

1.1 Transactional Memory

Transactional Memory (TM) was originally proposed as a hardware mechanism to simplify the creation of nonblocking data structures [1]. It then was embraced as a mechanism for lock elision [2], and has come to be seen today as a full-fledged programming model [3].

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 [4–6].

1.2 Condition Synchronization

There have been a handful of proposals for allowing the use of condition variables within transactions [5, 7, 8], but these have not been widely embraced. In contrast, the `Retry` mechanism [9] is a popular tool for coordinating transaction in Haskell. The distinction between condition variables and `Retry` is significant. Transactional condition variables break atomicity, by committing an in-flight transaction at the point of a `Wait`, and then running the remainder of the transaction as a new atomic region, after wakeup. In contrast, `Retry` casts condition 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, existing approaches to implementing `Retry` are complex and intimately tied to low-level details of an underlying software TM (STM) implementation [9, 10]. The mechanism operates 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 log 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, and hence all transactions appear to need to fall back to a high-overhead instrumented mode as soon as any transaction calls `Retry`.

we introduce a new mechanism for implementing `Retry`, which employs value-based validation [11, 12] to avoid overhead on in-flight hardware transactions, and to make the wakeup mechanism more precise (e.g., immune to false wakeups due to silent stores). We also show that our mechanism has broad utility: we use it to implement `Retry` in one HTM library and two STM libraries, we use it to implement the simpler `Await` mechanism for condition synchronization [13], and we construct a new predicate-based condition synchronization technique that we call $WaitPred$. We hope that the broad usefulness of this mechanism will encourage TM designers to begin supporting one (or all!) of these language-level condition synchronization constructs, so that programmers can gain more experience with coordinating transactions and ultimately provide case studies to the C++ TM specification effort.

Chapter 2

Condition Synchronization Mechanisms

2.1 RETRY

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 condition synchronization techniques. We first consider an implementation of `Retry` that is faithful in spirit to prior STM proposals for Haskell [9] and C++ [10].

Algorithm 1 assumes an eager STM with in-place updates, such as TinySTM [14], or the STM provided with GCC [15]. 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 updated directly.

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 re-execution is worthwhile.

The main challenge is to ensure that when a thread is put to sleep, its reads have not experienced a concurrent modification; otherwise it could miss its lone opportunity to be awoken. Assuming the underlying STM is opaque [16], the calling transaction has a consistent view of memory when `Retry` is called. Nonetheless, the transaction must ensure its reads remain valid while adding itself to the list of waiting threads. This necessitates some manner of validation atomic with the update to *waiting* (lines 3–8 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 good-faith 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. Traditionally, `Retry` performs 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 [17–19]. 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.

2.2 Deschedule - An abstract mechanism

2.2.1 A Motivating Example

Algorithm 2 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 has reasoned that since there are transactions, it ought to be possible to compose complex atomic behaviors involving the buffer. To this end, the programmer has crafted Algorithm 3, which illustrates a dangerous scenario of composing `Produce` and `Consume` methods. we name it $Produce1Consume2()$.

Suppose thread T calls $Produce1Consume2()$ when *count* is 0. Lines 1-5 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 6 is reached, the buffer is empty, and thus line 21 of algorithm 2 will be reached. To put T to sleep, the outermost transaction will commit, breaking atomicity. 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 of algorithm 2 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 much cleaner than condition variables. With our mechanisms, lines 15 and 23 of algorithm 2 are no longer necessary, and neither are the loops on algorithm 2's lines 10 and 18. The unrolling of a transaction when using our mechanisms provides an implicit back-edge.

Figure 2.1 illustrates the effective behavior of the mechanisms we discuss in this thesis. At time 1, $Waiter$ (executing $Produce1Consume2$) reaches an untenable state (algorithm 2 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 $WaitPred$ condition synchronization mechanism we introduce in this thesis, achieve this behavior.

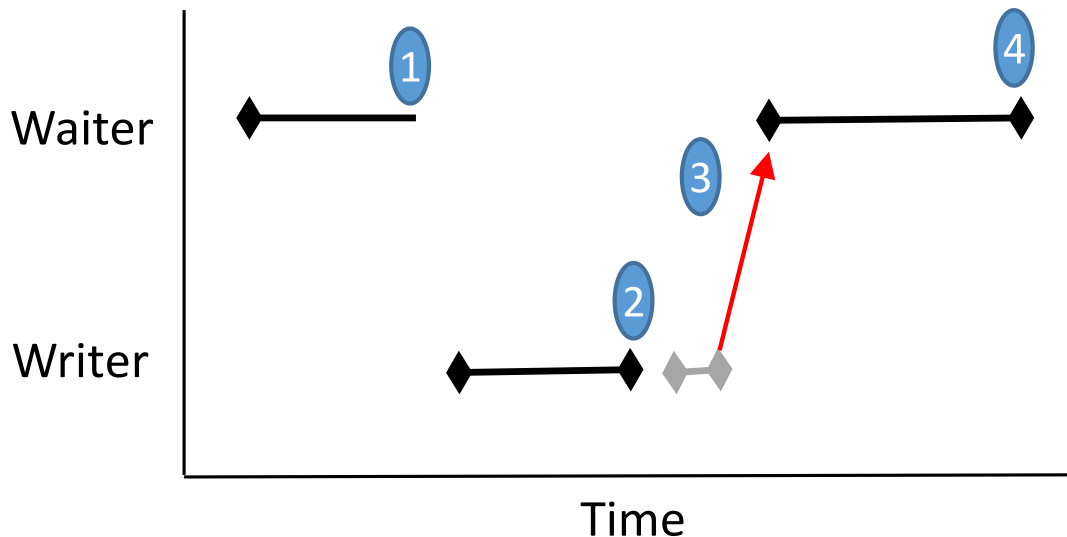


Figure 2.1: High-level interaction between a waiter and a writer. At time 2, the writer has changed the shared state in a way that makes re-attempting the waiter worthwhile. The decision to wake the waiter occurs at time 3.

2.2.2 An 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 2.1, 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 [20], 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.

Our technique (1) shifts overhead from the writer to the waiter, and (2) treats the wakeup operation as a computation over shared memory. Algorithm 4 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 4–8 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 9). 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 17).

2.2.3 Implementing RETRY

As discussed in Section 2.2.2, 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 5. 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`. 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 the $Deschedule(findChanges, self)$ method. That is, `Retry` will undo the transaction's effects, add the transaction to *waiting*, double-check that the values read by the

[1] Existing best-effort HTM implementations already require this fallback path to overcome transactional capacity limitations [5].

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*.

2.2.4 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 [13], 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` already available, it is relatively straightforward to also implement this limited interface, as depicted in Algorithm 6.

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" [21], 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 [22]), releasing locks would be incorrect.

Note that it is possible that addresses passed to *Await* were allocated by the transaction, as "Captured Memory" [23, 24]. 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.[2]

2.2.5 WaitPred: Synchronization with Explicit Predicates

Having developed support for `Retry` and *Await* through the use of *Deschedule*, we now have the means to add an additional mechanism for condition synchronization, which we call $WaitPred$.

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

[2]Strictly speaking, this concern is also possible in `Retry`.

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 7.

2.2.6 Discussion

There are 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. Note that we are also careful to avoid erroneous wakeups. For example, in *Await*, we explicitly do not store values into the *waitset* during $TxRead$, instead waiting until after the undo log has been rolled back to ascertain these values. 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 scheduling and condition synchronization as predicates over states, there are some unexpected scheduling outcomes. For example, suppose that 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 not a 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 go to sleep again. We contend that none of these outcomes are necessarily unintuitive, once the programmer is comfortable thinking of condition synchronization as predicates over program states (and indeed, this line of thinking originates with the original `Retry` mechanism). Since one outcome of our work is bringing `Retry` from its original home in Haskell to HTM, we believe that emphasizing this point is useful.

2.3 Programmability

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

```
procedure GetPred()
    TxBegin()
    if Empty() then
        WaitPred(¬Empty, void)
    return Get()
    TxCommit()
```

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

```
procedure GetAwait()
    TxBegin()
    if Empty() then
        Await(⟨&count⟩)
    return Get()
    TxCommit()
```

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

```
procedure GetRetry()
    TxBegin()
    if Empty() then
        Retry()
    return Get()
    TxCommit()
```

Figure 2.2: Put and Get methods for a bounded buffer using three transactional condition synchronization mechanisms.

Programming with *WaitPred* and *Await* are 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 2.2 show the changes to Listing 2 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.

An open question is whether this increased diversity of linguistic constructs for scheduling transactions will be valuable. There are three issues which we highlight in this section. First, the value of composition is not obvious. We conducted a survey of 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, they simply have not been adopted [25].

Second, we note that just as our mechanisms provide functionalities that are not available to condition variables (such as fine-grained control over which threads to wake up, via predicates), there are situations in which our mechanisms cannot be used as a simple replacement for condition variables. As a strawman, consider the implementation of a condition variable as an integer. To wait, one could simply use *Await*, passing the value of the integer, and to signal, one could increment the integer. In addition to the restriction that this would only provide broadcast functionality, we observe that this would not work for critical sections that expect to make their state updates visible to other threads. For example, the classic two-wait reusable barrier [26, Chapter 5] cannot be implemented via simple substitution.

This is a known problem [27], which we highlight to emphasize that some code must be re-designed when transitioning from condition variables to transactional mechanisms.

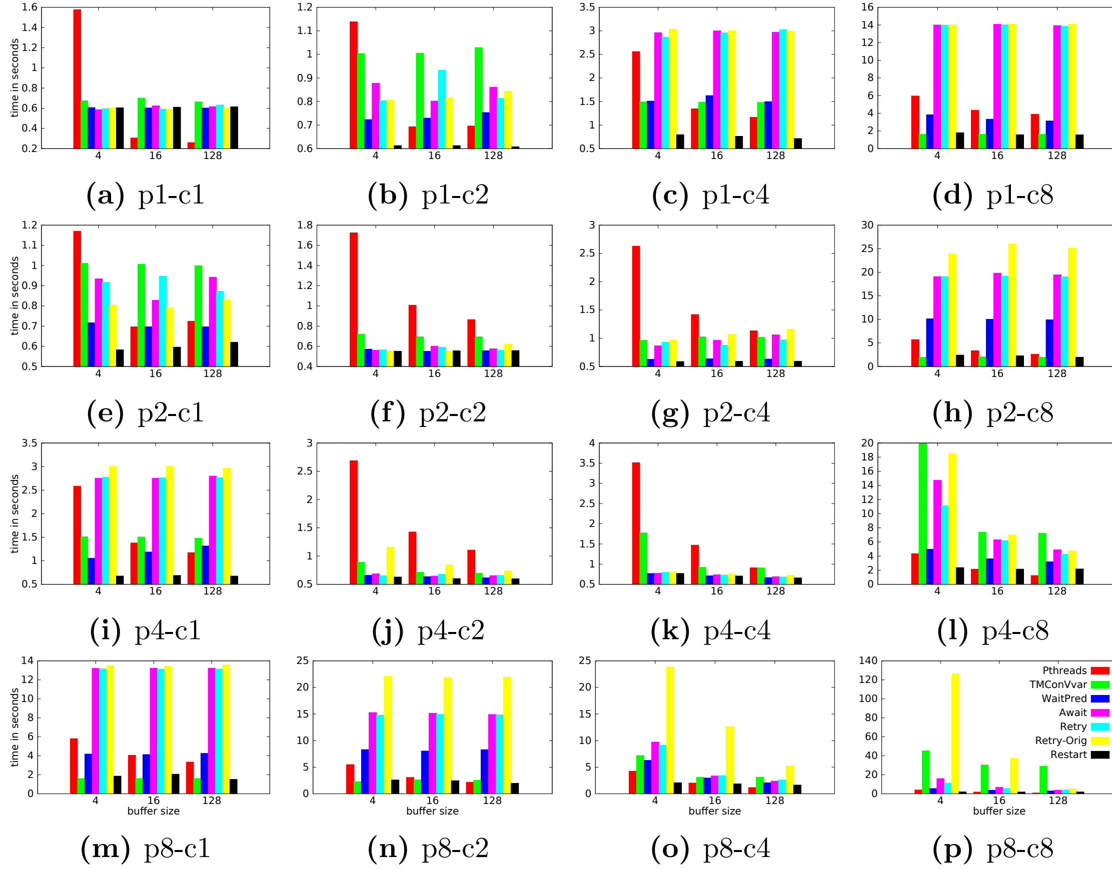
Finally, we observe that there is a tradeoff in the complexity of reasoning required when using $WaitPred$, $Await$, and $Retry$. Consider the $Produce1Consume2$ example from Section 2.2.1. In this case, the use of `Retry` within the Put and Get methods is sufficient, but the use of $WaitPred$ is not: 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 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.

Regarding this final point, the benefit of our mechanism is that it should introduce a tradeoff between the generality of the condition 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.

2.4 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)?



(a) p1-c1 **(b)** p1-c2 **(c)** p1-c4 **(d)** p1-c8

(e) p2-c1 **(f)** p2-c2 **(g)** p2-c4 **(h)** p2-c8

(i) p4-c1 **(j)** p4-c2 **(k)** p4-c4 **(l)** p4-c8

(m) p8-c1 **(n)** p8-c2 **(o)** p8-c4 **(p)** p8-c8

Figure 2.3: Bounded buffer performance with eager STM.

- 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, and where there may be more threads than cores. Second, we measure performance on the PARSEC benchmark suite [28]. We limit our evaluation to the 8 PARSEC benchmarks that use condition variables.

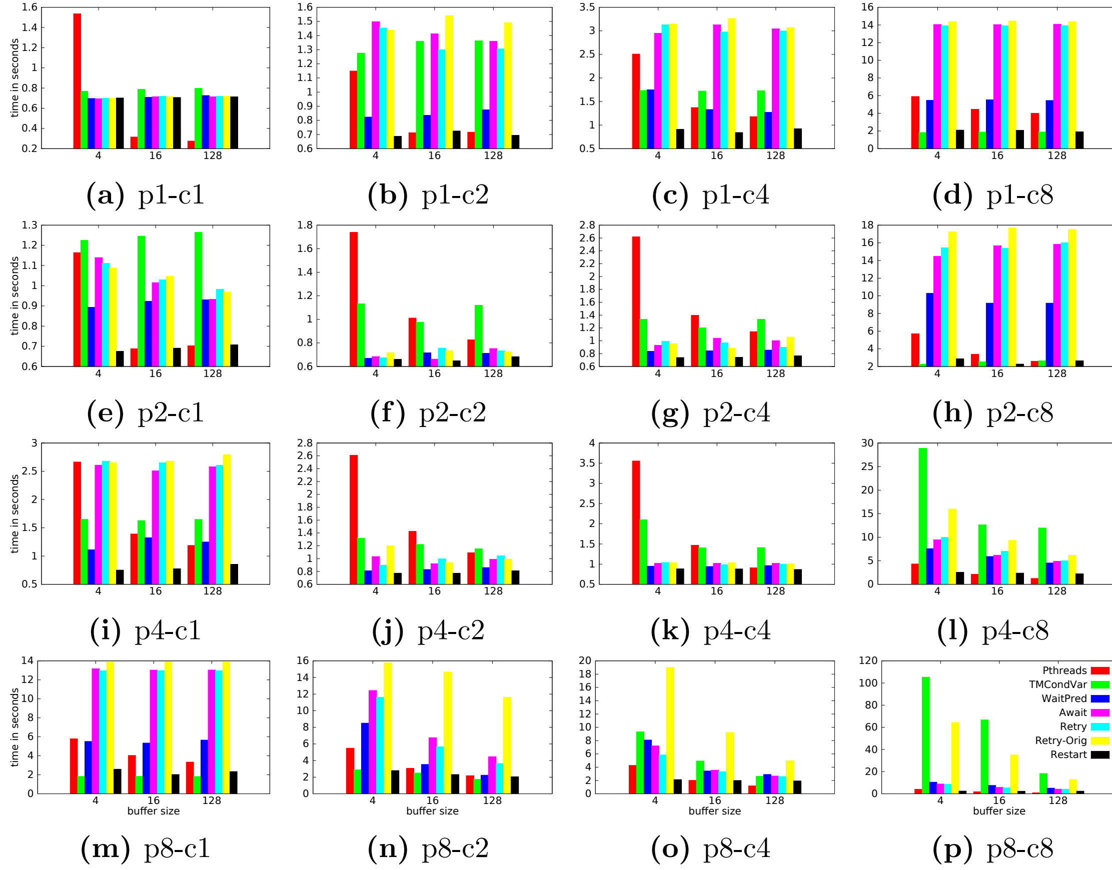


Figure 2.4: Bounded buffer performance with lazy STM.

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 [7] 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 [9]. Finally,

Restart aborts and immediately restarts a transaction any time a precondition does not hold.

In our experiments, we 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 [14]). Lazy STM is like Eager STM, but uses redo logs, much like a privatization-safe version of TL2 [29]. 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.

2.4.1 Producer Consumer Micro-benchmark

We begin with experiments on a bounded buffer micro-benchmark, based on the Figures 2.2. 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 starting each experiment.

Figures 2.3–2.5 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, p2c2, p4c4), our mechanisms have the best performance: they avoid calls into the pthread library, and when there are wakeups, only a small number of threads are woken at a time. The effect is most pronounced for small buffers, where sleeping is more likely. Furthermore, our mechanisms 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 (p2c1, p4c2, 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 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 moderate imbalance (p4c1, p1c4, p8c1, p8c2, p1c8, 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.

For HTM, high imbalance experiments (p8c4, p4c8, p8c8) behave the same as under moderate 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

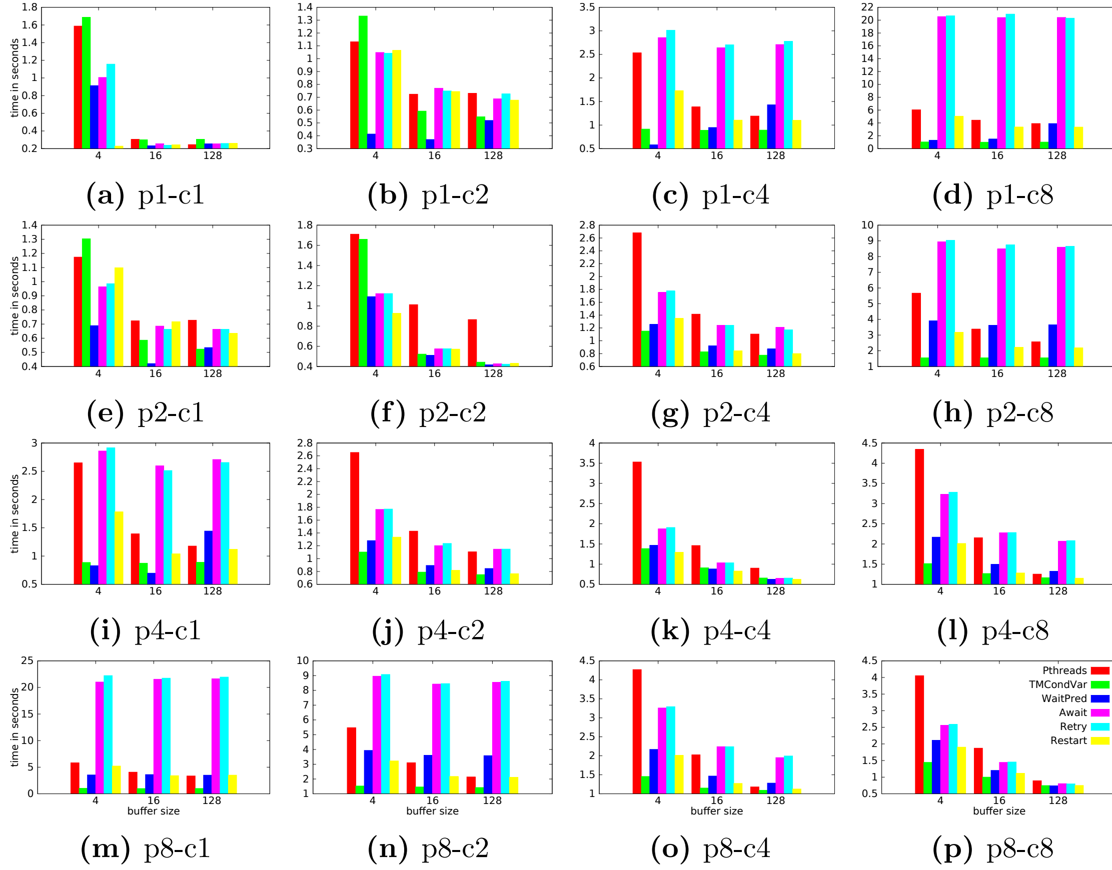


Figure 2.5: Bounded buffer performance with HTM.

3× latency overhead of STM instrumentation dominates, and variance also increases to above 1.

2.4.2 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 2.1 describes the number of lines of code that were removed from each benchmark to eliminate condition variables, and the number of lines that

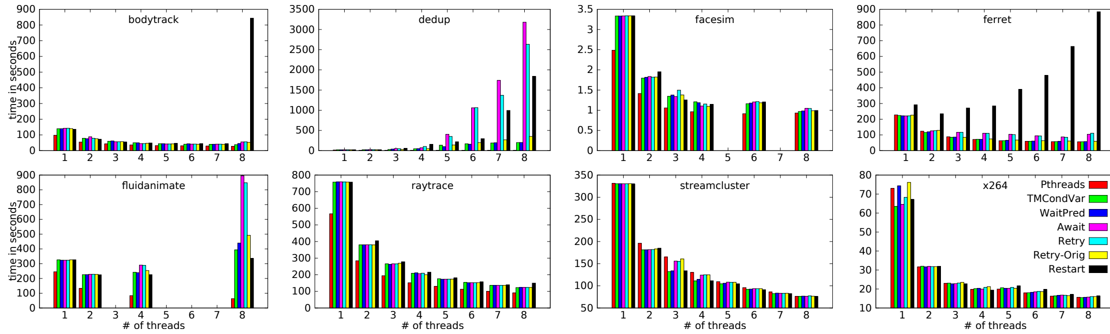


Figure 2.6: PARSEC performance with eager STM.

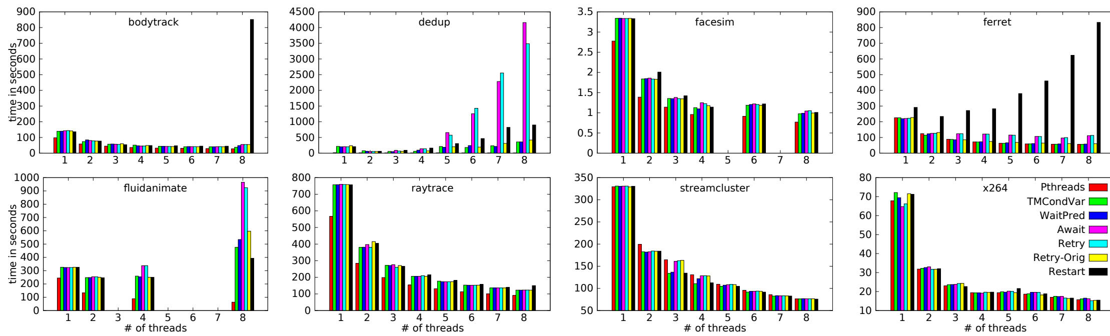


Figure 2.7: PARSEC performance with lazy STM.

were added to synchronize threads via our mechanisms.

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 2.6–2.8 present PARSEC performance for STM and HTM. Each bar

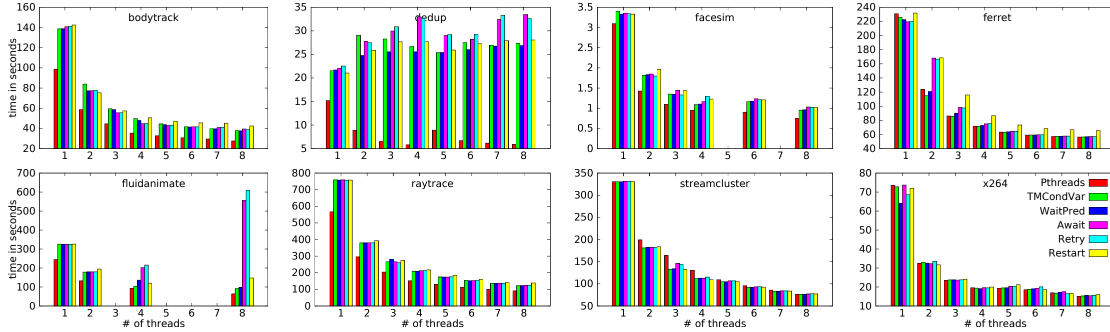


Figure 2.8: PARSEC performance with HTM.

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. [7]. 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 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, *Await* tends to *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 [3], 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.

2.5 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 [30] and Yoo et al. [6]. Dudnick and Swift subsequently presented hardware and OS extensions that allowed for transaction-safe condition variables [8], and Yoo et al. later showed that existing hardware could be made compatible with condition variables through a novel use of Linux futexes [5]. Wang et al. subsequently proposed an OS-agnostic, hardware-agnostic mechanism for transaction-safe condition variables, but it required extensions to the compiler [7]. 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 [31], and later the X10 group [32], 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 [9], which we study in this paper. Our work completely separates the retrying mechanism from the underlying TM implementation, and offers programmer control over the expression of the precondition. Thus one benefit of our work is making it possible to use `Retry` in HTM and Hybrid TM.

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 [27]. 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 reestablish atomicity. Proposals for synchronizing transactions via group commit were proposed by Luchangco and Marathe [33] and Lesani and Palsberg [34]. Luchangco later showed that this technique can approximate condition variables [35]. However, the techniques have high complexity and are not compatible with HTM [36].

Algorithm 1: Original Retry mechanism, adapted from [9] to use eager STM. A lock prevents concurrent accesses for simplicity.

```
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 waiting.unlock()
      // restart the transaction
 9    reads ← undos ← locks ← {}
10    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 ← {}
```

Algorithm 2: A bounded buffer example using transactional CondVars

```
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()
```

Algorithm 3: A dangerous Scenario

```
    procedure Produce1Consume2(x)
1       TxBegin()
2       inprogress ← true
3       x ← CreateElement()
4       Produce(x)
5       a ← Consume()
6       b ← Consume()
7       Use(a, b)
8       inprogress ← false
9       TxCommit()
```

Algorithm 4: 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 predicate (f) and it's record of paramters p
   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(); waiters ← waiters − self; TxCommit()
       // restart the parent transaction
17     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; l ← waiting.copy(); TxCommit
       // check each entry's condition
 2     for e ∈ l do
 3         shouldWake ← false
 4         TxBegin
 5         if e.asleep ∧ e.waitfunc(e) then
 6             e.asleep ← false
 7             shouldWake ← true
 8         TxnCommit
 9         if shouldWake then  e.semaphore.signal()
```

Algorithm 5: 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)
```

Algorithm 6: 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 7: 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)
```

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

Chapter 3

Conclusion and Future Work

We introduced a new approach to transactional condition synchronization. Our algorithms are inspired by `Retry`, but offer two powerful new abilities: First, the programmer can fine-tune the condition upon which a transaction depends, instead of tracking memory locations. Second, our mechanisms are compatible with HTM, Hybrid TM, and STM implementations.

Our evaluation showed that our multiple linguistic constructs, supported by a single implementation, had minimal impact on code size, and that their performance impact on large benchmarks like PARSEC is negligible. On stress-test micro-benchmarks, performance is more nuanced, but overall, it appears that our approach simultaneously achieves the goals of ease-of-use, performance, generality, and amenability to programmer optimization/tuning.

The most significant open issue with our work is studying the relationship between condition synchronization 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 or not is an open question, as is the question of whether such compatibility is sufficient. We are comforted by our experience with PARSEC, but encourage further workload and application usage studies.

Bibliography

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

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

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

[4] Ruan, W., Vyas, T., Liu, Y. & Spear, M. 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, 2014).

[5] Yoo, R., Hughes, C., Lai, K. & Rajwar, R. 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, 2013).

[6] Yoo, R. *et al.* 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, 2008).

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

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

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

[10] Spear, M., Dalessandro, L., Marathe, V. J. & Scott, M. L. 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, 2009).

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

[12] Olszewski, M., Cutler, J. & Steffan, J. G. 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, 2007).

[13] Carlstrom, B. D. *et al.* The Atomos Transactional Programming Language. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation* (2006).

[14] Felber, P., Fetzer, C. & Riegel, T. 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, 2008).

[15] Free Software Foundation. Transactional Memory in GCC (2012). Http://gcc.gnu.org/wiki/TransactionalMemory.

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

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

[18] Dalessandro, L. *et al.* 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, 2011).

[19] Matveev, A. & Shavit, N. 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, 2015).

[20] Zilles, C. & Baugh, L. 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, 2006).

[21] Ni, Y. *et al.* 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, 2008).

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

[23] Riegel, T., Fetzer, C. & Felber, P. Automatic Data Partitioning in Software Transactional Memories. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures* (Munich, Germany, 2008).

[24] Dragojevic, A., Ni, Y. & Adl-Tabatabai, A.-R. Optimizing Transactions for

Captured Memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada, 2009).

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

[26] Anderson, T. & Dahlin, M. *Operating Systems Principles & Practice* (Recursive Books, 2014).

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

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

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

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

[31] Harris, T. & Fraser, K. Language Support for Lightweight Transactions. In

Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (2003).

[32] Charles, P. *et al.* 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, 2005).

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

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

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

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

[37] Menon, V. *et al.* Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures* (Munich, Germany, 2008).

[38] Spear, M., Marathe, V., Dalessandro, L. & Scott, M. Privatization Techniques for Software Transactional Memory (POSTER). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing* (Portland, OR, 2007).

Appendix A

Software TM Implementation Details

In this appendix, we present an implementation of a software TM that uses undo logs, similar to TinySTM [14]. This presentation provides background that is useful for understanding the behavior of our *Deschedule* algorithms.

Algorithm 8 presents the variables and data types. A set of locks protects all shared memory, and a hash function is used to map memory addresses to locks. We assume that it is possible to atomically read all fields of a Lock object simultaneously, and to modify Lock objects via an atomic compare-and-swap (CAS) instruction. As in TL2 [29], a monotonically increasing clock is incremented on each writer transaction commit. This significantly reduces the overhead of validating that transactional reads are consistent.

A Tx object is associated with each thread. The Tx object contains sets for undoing writes, tracking locks held, and tracking locations read. There is also

Algorithm 8: Global and thread-local variables for a software TM implementation. The "*" suffix indicates set types.

Globals			
	locks	: $Lock^*$	// Set of locks
	clock	: $Integer$	// Logical clock [29] to count commits
Fields of the Lock Object	*locked*	: $Boolean$	// True iff lock is held
	owner	: Tx	// Identity of lock holder
	version	: $Integer$	// Time of last unlock
Fields of the Tx Object	*undos*	: $\langle addr, val\rangle^*$	// Writes by this transaction
	locks	: $Lock^*$	// Locks held by this transaction
	mallocs	: $addr^*$	// Result of calls to `malloc`
	frees	: $addr^*$	// Deferred calls to `free`
	reads	: $addr^*$	// Reads by this transaction
	nesting	: $Integer$	// (Flat) nesting depth
	cp	: $Checkpoint$	// Used on abort/rollback
	start	: $Integer$	// Time of transaction start

metadata for managing a thread's rollback, and for handling nested transactions via flat (subsumption) nesting. Additionally, there are sets for deferring reclamation and undoing allocations.

To begin an new lexically scoped transaction (Algorithm 9), a thread increments its nesting counter. If the counter was not zero, then a nested transaction is started, and no further work is required. Otherwise, the thread creates a checkpoint, so that aborted transaction attempts can restore the architectural state to precisely as it was when $TxBegin$ was called. It also reads the current value of the *clock*, so it can easily identify locations that are safe to access (e.g., those whose last modification preceded this transaction's start).

$TxWrite$ is called on any write of shared memory (Algorithm 10). To write shared memory, the transaction must hold an exclusive lock over the to-be-written address. If such a lock is not yet held, the transaction must atomically transition the location's lock from a state in which it is unlocked and no newer than the

Algorithm 9: Begin and end instrumentation for a simple software TM

```
procedure TxBegin
    // create checkpoint iff outermost transaction
1   nesting ← 1 + nesting
2   if nesting = 1 then
3       cp ← createCheckpoint()
4       start ← clock

procedure TxCommit
    // handle nesting
1   nesting ← nesting − 1
2   if nesting > 0 then
3       return
    // handle read-only transactions
4   if locks.isEmpty() then
5       reads ← {}
6       return
    // get commit time
7   end ← atomicIncrement(clock)
    // validate
8   if end ≠ start + 1 then
9       for addr ∈ reads do
10          tmp ← locks[hash(addr)]
11          if ¬tmp.locked ∧ tmp.version > start then
12              TxAbort()
13          if tmp.locked ∧ tmp.owner ≠ me then
14              TxAbort()
    // transaction is committed... release locks
15  for l ∈ locks do
16      l ← ⟨false, nil, end⟩
    // finalize frees
17  for f ∈ frees do
18      free(f)
    // reset lists
19  reads ← undos ← locks ← mallocs ← frees ← {}
    // quiesce to ensure privatization safety
20  quiesce()
```

transaction's start time, to a state in which it is locked by the transaction. If this attempt fails, the transaction aborts. Once the lock is held, the transaction copies

Algorithm 10: Read and write instrumentation for a simple software TM

```
    procedure TxWrite(addr, val)
        // Atomically read the lock object
1       tmp ← locks[hash(addr)]
        // Handle locks already held by caller
2       if tmp.locked ∧ tmp.owner = me then
            // add old value to undo log, update, return undos.append(⟨addr, *addr⟩)
4           *addr ← val
5           return
        // Only succeed if lock version not too new
6       if ¬tmp.locked ∧ tmp.version ≤ start then
7           if CAS(locks[hash(addr)], tmp, ⟨true, me, tmp.version⟩) then
                // same process as above, but also record this lock
8               locks ← locks ∪ locks[hash(addr)] undos.append(⟨addr, *addr⟩)
10              *addr ← val
11              return
        // In all other cases, abort
12      TxAbort()

    function TxRead(addr)
        // Atomically read the lock object
1       tmp ← locks[hash(addr)]
        // read the location, then re-check the lock object
2       val ← *addr
3       tmp2 ← locks[hash(addr)]
        // Easy case: caller holds lock
4       if tmp.locked ∧ tmp.owner = me then
5           return val
        // Only succeed if read is consistent
6       if tmp = tmp2 ∧ ¬tmp.locked ∧ tmp.version ≤ start then
7           reads.append(addr)
8           return val
        // In all other cases, abort
9       TxAbort()
```

the old value at the location into the undo log, and then updates the location. Note that this copy is required even on line 3, since a single lock can cover multiple locations.

Whenever a location is read, the *TxRead* instrumentation is called. If the location is already locked by the caller, then the location can simply be read (line 5).

Otherwise, the transaction must atomically read the lock and location, then ensure that the lock is in a safe state for this transaction (i.e., it is unlocked and its version is no greater than the caller's start time). When a location is read, its address is added to the read set, so that we can ensure that all reads are valid when the transaction commits.

To commit a transaction, $TxCommit$ first checks if the transaction is nested. If so, no work is needed, as the parent transaction is not yet complete. Next, read-only transactions are handled. To reach $TxCommit$, every read by the transaction must have passed the test on line 6 of $TxRead$, and hence all values that were read were logically present in memory at the time when the transaction started. Thus no further processing is required. Otherwise, the transaction must validate. This ensures that all reads were valid immediately after the time at which the last lock was acquired. A fast-path is used (line 8) to detect when no other transactions committed between this transaction's begin and end. The validation ensures that every read location is either (a) unlocked and not updated since this transaction started, or (b) locked by this transaction. Since this transaction only acquired locations that were not written after it began ($TxWrite$ line 6), read-then-write accesses are not a concern. If the validation succeeds, the transaction releases its locks and updates their version number. It then performs any deferred reclamation, and resets its metadata. Finally, it uses a quiescence technique [37] to ensure privatization safety [38].

Algorithm 11 presents the abort code for a transaction. $TxAbort$ is responsible for undoing writes, releasing locks, and resetting the transaction's metadata. Then it restores the checkpoint, so the transaction may restart. There are two subtleties. First, care is required when releasing locks, to ensure that a concurrent $TxRead$

Algorithm 11: Abort routine for software TM

```
procedure TxAbort
       // undo all writes
1      for ⟨addr, val⟩ ∈ undos.reverse() do
2          *addr ← val
       // release all locks; increment to notify TxRead lines 2-3
3      for l ∈ lock_set do
4          l ← ⟨false, nil, l.version + 1⟩
       // ensure released locks have legal versions
5      atomicIncrement(clock)
       // undo allocations by this transaction
6      for m ∈ mallocs do
7          free(m)
       // reset lists
8      reads ← undos ← locks ← mallocs ← frees ← {}
       // re-start from beginning of transaction
9      nesting ← 0
10     cp.restore()
```

does not observe out-of-thin-air values. Secondly, any allocations performed by the transaction must be undone.

The mechanics of allocating and freeing memory are not shown in pseudocode; they are straightforward. Calls to $free()$ are replaced with calls that insert the to-be-freed pointer into the $frees$ collection. Calls to $malloc()$ are replaced with calls to a function that performs a $malloc$ and then inserts the return value into $mallocs$. When a transaction commits, $mallocs$ is cleared and the entries in $frees$ are freed. When a transaction is aborted, $frees$ is cleared and the entries in $mallocs$ are freed. In this manner, reclamation is delayed until it is certain that the reclamation need not be rolled back, and allocations can be logically undone if the calling transaction aborts.

We observe that the above implementation is not optimal, but it is sufficient to illustrate the behavior of software TM. There are two key aspects in which the

implementation deviates from the implementation in GCC. First, our decision to abort upon encountering “too new” version numbers in $TxRead$ and $TxWrite$ is overly conservative, and can be avoided via timestamp extension techniques [22]. Second, our blind increment of the clock in $TxAbort$ line 5 can be avoided through the use of incarnation numbers [14].

Vita

The author, Chao Wang, was born on Dec 14th, 1987, in Jincheng, Shanxi Province, China. His father is Yonglu Wang and his mother is Suxia Huo. He graduated from Beijing University of Posts and Telecommunications, China with a Bachelor of Engineering degree in July, 2009. Then he went to Institute of Computing technology, Chinese Academic of Sciences. He received his Master of Science degree there in June, 2012 before he came to Lehigh University. During he stay at Lehigh University, he served as a research assistant with Professor Michael Spear and teaching assistant for a course named Programming Languages. His research interest lies in parellel/concurrent programming, machine learning etc.