A New API For Transactional Condition Synchronization *

Chao Wang
Lehigh University
chw412@lehigh.edu

Yujie Liu
Lehigh University
lyj@lehigh.edu

Michael Spear
Lehigh University
spear@cse.lehigh.edu

Abstract

In this paper, we introduce a new condition synchronization mechanism based on the idea of predicates and states. Our algorithm is compatible with hardware, software, and hybrid transactional memories, is simpler to implement than the current state of the art, and does not rely on extensive support from run-time libraries.

1. Introduction

Efficient support for transactional condition synchronization has been a difficult issue for many years [8]. The most straightforward solution is to make condition variables "safe" for use in transactions, by using OS-specific mechanisms [7], new hardware features [2], or custom runtime support [6] for non-lexically-scoped transactions. In these approaches, a transaction that waits on a condition is split into two different transactions [5].

While these solutions are sufficient for transactionalizing legacy code, they preserve a cumbersome programming model: Since they entail breaking a single atomic transaction into two separate atomic sections (one before, and one after, the wait on the condition variable), they require whole-program reasoning about the composition of transactions that must wait. A much more appealing approach is to think of condition synchronization as an approach to scheduling transactions: If a transaction can proceed only if a certain precondition hold, then the transaction should not execute until some other thread establishes the precondition.

The simplest compatible programming model is to guard certain transactions via a read-only prefix, in a manner reminiscent of conditional critical regions [1, 3]. The transaction can execute the read-only prefix and determine whether the precondition is met; if so, it continues to execute, and otherwise it aborts, waits, and then tries again. To make waiting efficient, the aborting transaction can put a description of the locations it read into a shared data structure when putting its thread to sleep, and then any transaction that commits changes to any of those locations must wake the waiting thread. To support nesting and composition, Harris et al. proposed a linguistic construct called `retry`. When `retry` is called within a dynamic (and possibly nested) transactional context, the read set of the transaction is made visible to future transactions, the speculative writes of the transaction are discarded, and the transaction is put to sleep until some other transaction modifies a location that the `retry`ing transaction read [4].

We observe that there are two problems with `retry`: First, it is imprecise in that any store to any location read by a waiting transaction will cause the waiting transaction to wake. This includes stores that do not establish the required precondition, even silent stores. Second, `retry` requires the read and write sets of all transactions to be visible to the runtime library. This precludes the use of `retry` in hybrid transactional memory (HyTM) systems, where the read and write sets of transactions may be known only to the hardware.

In this paper, we introduce a new condition synchronization mechanism called TXNRESCHEDULE, which leverages explicit predicates specified by the programmer. To wait on some precondition, a thread publishes a predicate in the form of a lambda expression, evaluates the precondition and puts itself to sleep if the condition is not satisfied. Assuming that the transactional memory (TM) implementation ensures a total order on transactions, the lambda can be regarded as an atomic evaluation of the given predicate on the shared state at a time when no other transactions are executing. If that predicate evaluates to true, then it is possible that the

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

waiting transaction could complete successfully if re-executed.

To wake a thread, a transaction evaluates every published lambda *after* it commits. If a lambda returns true, then either the calling thread or some concurrent thread has just completed a transaction that establishes the precondition upon which the waiting thread's transaction depends. Therefore, it is acceptable to wake the thread. Since intermediate effects of transactions are not visible, spurious wakeups do not compromise correctness. Additionally, since the evaluation of lambdas follows the commit of a transaction, that transaction can employ hardware TM resources when available. The result is a system that can maximally use hardware TM, and that allows condition synchronization based on a composable mechanism (unwinding and delaying the execution of transactions, instead of committing partial results and breaking atomicity [6]).

2. An Example

Figure 1 gives an intuition into the behavior we are proposing for condition synchronization among transactions. In step 1, Thread T_1's transaction cannot complete, due to some precondition not holding. T_1 rolls back its effects and adds a lambda (function + parameters) into a shared set. T_2 similarly cannot complete (step 2), so it enqueues a different lambda. After T_3 completes (step 3), it uses separate transactions to evaluate the two lambdas published by T_1 and T_2 (steps 4 and 5). The computation for T_1's lambda returns true, so T_1 resumes and the lambda is removed from the set. T_2's lambda does not return true, so T_2 continues to sleep. After T_1's transaction commits (step 6), there is one lambda in the set, so T_1 uses a transaction to evaluate the lambda (step 7). This time, the result is true, so T_2 is woken and the lambda removed from the set. When T_2 commits its transaction (step 8), the set is empty, so no additional processing is required.

For the same workload, the behavior of `retry` would be as follows: In steps 1 and 2, T_1 and T_2 would store in a shared set a description of all locations they read during their transactions. During step 3, T_3 must save a description of all locations it wrote during its transaction, and then steps 4 and 5 would consist of computing intersections between T_3's "write" description and the "read" descriptions from T_1 and T_2. From a programmability perspective, `retry` is *easier*, as it does not require the programmer to encode the state

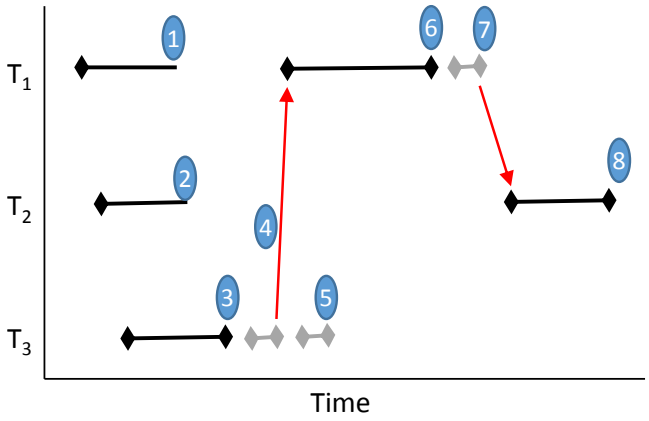

Figure 1: Example interaction between three threads.

upon which T_1 or T_2 depends: *any* change to *any* location read by either waiting transaction will cause the corresponding thread to resume. However, from an performance perspective, our mechanism is more desirable: since the read and write sets of transactions need not be visible to the run-time system, it is possible to use hardware TM to execute T_3's transaction, the subsequent transactions by T_1 and T_2, and all transactions for evaluating lambdas. Similarly, in some programs it is possible to execute via HTM those transactions that ultimately must wait (see Section 4). Note, too, that our mechanism can be used to simulate `retry`, by encoding an explicit validation of address/value pairs as the lambda for the waiting transaction.

3. A Generic Algorithm

We present a generic algorithm for rescheduling transactions in Algorithm 1. We add one method to the TM API: TXNRESCHEDULE, which a transaction can use to roll back its attempt and then delay its re-execution until some other transaction establishes the necessary precondition. We use AFTERTXNCOMMIT to describe code that should be executed by a thread after it successfully commits a transaction.

We assume that a transaction's abort mechanism consists of two parts: rolling back its changes to shared state and per-thread metadata (RESETMETADATA) and restoring a checkpoint in order to re-attempt the transaction (ROLLBACK). To abort a transaction and delay its resumption until after some predicate is established, TXNRESCHEDULE atomically checks the condition upon which it depends, and if the check fails, it adds its caller to the set of waiting threads (Q). The function does not return until some other thread removes

Algorithm 1: A Generic Algorithm for TM-Reschedule

```
   shared states
     Q    : Set<Thread>    // waiting threads; initially ∅
   // p refers to the thread that performs the operation
   // c refers to a wait condition (function and parameters)
   // t refers to the transaction p is executing
   procedure TxnReschedule(c)
1      t.ResetMetadata()
       // lines 2–4 execute atomically
2      if ¬Eval(c) then
3          p.c ← c
4          Q ← Q ∪ {p}
       // the test on line 5 executes atomically
5      while p ∈ Q do
           wait()
6      t.Rollback()
   procedure AfterTxnCommit()
       // line 7 executes atomically
7      Set<Thread> l ← Q.elements
8      for x ∈ l do
           // lines 9–10 execute atomically
9          if x ∈ Q ∧ Eval(x.c) then
10             Q ← Q \ {x}
```

the caller from Q. Note that it is necessary to test the condition atomically with the addition of the caller to Q, but that this need not be atomic with respect to the reads and writes performed by the aborted transaction. Note too that in a practical system, the condition (c) must be passed by value, not reference, or else the reset on line 1 might undo changes to c. We assume that c has no side effects.

The mechanism to wake threads is invoked immediately after any transaction commits, and is presented as AfterTxnCommit.[1] The committing transaction copies the list of waiting transactions, so as to avoid interaction with concurrent waiting and committing threads. Then, for each entry in the copy, the thread tests the associated condition, and if the condition holds, then the waiting thread is removed from the set, so that it may resume.

4. A Practical Algorithm

Our implementation of TxnReschedule assumes that a software TM runtime is available, at least, as a fall back option. In hardware/hybrid TM environments, calls to TxnReschedule from a hardware transaction are likely to cause the transaction to restart in software mode, although some exceptions are discussed at the end of this section.

There are three challenges when attempting to achieve a practical implementation of Algorithm 1. First, the algorithm relies upon spin waiting, which can waste processor cycles in the event that threads must wait for an extended period of time. This is easy to solve by employing per-thread semaphores, as in our prior work on transactional condition variables [6]. The solution entails assigning a semaphore to each thread as sem_p, replacing line 5 with $sem_p.wait()$, and then adding a line 11, $sem_x.signal()$. This line should run only if the condition on line 9 is true, but should not be atomic with respect to lines 9–10.

The second challenge is that we have not described how to achieve atomicity throughout the algorithm. A naive approach would employ 4 transactions. Using semaphores would eliminate the need for a transaction on line 5, and the transactions on lines 7 and 9–10 are not concerning, since the calling thread is outside of a transaction. However, we still require one transaction (lines 2–4) that must run while its caller (the about-to-wait transaction) still has an active checkpoint. If the runtime system allows open nested transactions, then lines 2–4 can run as an open nested transaction. Otherwise, some careful design is required.

Since the execution of Eval(c) may access data that is simultaneously being accessed by concurrent transactions, we must use transactions to achieve atomicity for this code. We briefly present two interesting implementation options. First, we could save the caller's checkpoint to a local variable before line 2, and then put it back into the caller's transaction metadata after line 5. Note that the call to t.ResetMetadata would have cleared all other per-transaction metadata, so that the checkpoint was the only remaining artifact of the aborting transaction. Thus using a transaction to execute lines 2–4 would be possible.

The second option leverages the fact that when line 2 is reached, the calling context is effectively a software transaction that has performed no reads or writes. That being the case, we could execute lines 2–4 within the context of the calling transaction. If they lead to an abort (which we expect to be unlikely since predicates should be small and accesses to the potentially

[1] Strictly speaking, read-only transactions need not call this method.

contended set Q occur only at the end of the region), the calling transaction will restart. This is an acceptable outcome, since the calling transaction may discover that the contention when attempting EVAL(c) was due to some other transaction establishing the desired invariant. If line 4 completes, the calling transaction can commit but not discard its checkpoint. Doing so would lead to all per-thread metadata being re-cleared, and the caller's checkpoint remaining valid for use on line 6.

The third challenge in achieving a practical implementation is that under most circumstances, a call to TXNRESCHEDULE that is made by a hardware transaction must cause the transaction to restart as a software transaction. When there are more threads than cores, this results in a delay before scheduling some other thread that might be able to make progress. Given that Intel's hardware transactional memory support allows the programmer to emit an 8-bit value to describe any explicit self-abort, there is a workaround for a limited class of possible predicate functions EVAL(c). Recall that a predicate in our proposal consists of a function and its by-value parameters. For a given program, if there are less than 256 unique combinations of predicate and parameters, then it is possible to produce a table containing all possibilities, and then allow a hardware transaction to reschedule by (1) self-aborting with the desired status code as the parameter to the HTM abort function, (2) looking up the appropriate function EVAL(c) in the handler for self aborts, (3) executing the equivalent of lines 2–5, and then (4) resuming the transaction from the abort handler.

5. Conclusions and Future Work

In this paper we introduced a new mechanism for transactional condition synchronization. Unlike prior work, our algorithms do not require splitting atomic transactions, and do not forbid the use of hardware transactional memory for non-waiting transactions. Given these properties, we are hopeful that our mechanism will provide an appealing alternative to both the unappealing programming model of transactional condition variables and the unappealing performance constraints of the `retry` mechanism.

As future work, we plan to implement our algorithms and perform a wide range of case studies to determine the utility of TXNRESCHEDULE, both when rewriting legacy code to use transactions, and when creating new programs from scratch.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.