Model checking transactional memory with Spin

John O'Leary
Intel
john.w.oleary@intel.com

Bratin Saha
Intel
bratin.saha@intel.com

Mark R. Tuttle
Intel
tuttle@acm.org

Abstract

We used the Spin model checker to show that Intel's implementation of software transactional memory is correct. Transactional memory makes it possible to write properly-synchronized multi-threaded programs without the explicit use of locks. We describe our model of Intel's implementation, our experience with Spin, what we have shown, and what obstacles remain to showing more.

1. Introduction

Transactional memory [17], [36], [26] is a programming abstraction that makes it possible to write properly-synchronized multi-threaded programs without the explicit use of locks, and without the problems that come with locks [17]. Transactional memory provides a construct `atomic{block}`, where `block` is any block of program code, that makes the execution of `block` appear atomic relative to individual steps of other threads in the program, much like transactions in a database system.

The *privatization problem* [26, p. 22] illustrated in Figure 1 shows how tricky transactional memory implementations can be. Thread A atomically removes the element at the head of a list, and then reads the value of this element three times. Thread B atomically increments the value of every element in the list.

From the programmer's point of view, this code is elegant and properly synchronized. Thread A can safely read the value of the removed element outside the atomic block because A and B cannot access this element concurrently. Either A removes this element before B has a chance to increment its value, or B increments its value before A removes the element. In either case, B will never attempt to access the element after A has removed it.

From the implementer's point of view, however, care is required. Suppose the threads are allowed to update the lists in place [1], [15], and consider the following execution. Thread B starts executing and stores the pointer to the head of the list in a local variable. Thread A removes the head of the list and reads the value 0 from the former head. Thread B then uses the stale pointer to the former head of the list and increments every list element. Thread A reads the value 1 from the former head. Thread B realizes that the list has changed, and aborts after undoing its effects and resetting the list elements to 0. Finally, thread A reads the value 0 from the former head. At this point, A has made a private copy of the head of the list, yet has seen the value of this private copy change twice from 0 to 1 to 0.

What went wrong? Who should the specification of transactional memory blame for this unexpected behavior? Is the implementation wrong because an illegal (intermediate) state was seen, or is the program wrong because a memory location was accessed both inside and outside an atomic block? This is the kind of question we would like to be able to answer. We want to specify the correctness of transactional memory, and then to prove the correctness of transactional memory implementations.

This paper describes our verification of Intel's implementation of software transactional memory [34]. This implementation goes by the name *McRT STM* because it is part of an experimental Many-core Run Time environment for terascale computing being built at Intel [33]. McRT STM was released to the general public at the Intel Developer Forum in 2007 and binaries are available for download at whatif.intel.com.

This paper makes two contributions:

1) **The validation of an industrial implementation of transactional memory.** We used the Spin model checker [18] to prove the following: McRT STM guarantees the serializability of every execution of every purely-transactional program consisting of two threads each running one transaction consisting of three reads or writes. (An execution is serializable if transactions appear to be executed without overlap in a sequential order, and an execution is purely-transactional if every read and write occurs within a transaction.) Our model of McRT STM is very close to the C++ code used in the actual McRT STM implementation, in contrast to other applications of model checking to transactional memory that use very abstract models of the implementations. Our model of programs running on top of McRT STM is very efficient, in that checking the correctness of just a single program described above takes almost a minute (45 seconds), but we can check the correctness of all such programs (tens of thousands) in just over an hour (1:06).

```
A: /* remove list head */
   atomic {
    result := head;
    if head != null then
      head := head.next;
  }
  a := result.value;
  b := result.value;
  c := result.value;
```

```
B: /* increment list elements */
   atomic {
    node := head;
    while (node != null) {
      node.value++;
      node := node.head;
    }
```

Figure 1. The privatization problem.

2) **A tool to optimize models of shared memory algorithms in Spin.** We have gained a deeper understanding of how to model shared memory algorithms in Spin, and we have built a Spin preprocessor `spp` that captures some of this insight. The preprocessor rewrites our Spin code in a way that lets the Spin compiler apply its optimizations more effectively and dramatically improves the performance of the resulting model checking task. The preprocessor allows us to write the fully-parametrized models any good computer scientist would be inclined to write — with parameters for the number of threads, length of transactions, number of memory locations, etc. — but to generate for any particular instantiation of these parameters a model that runs much faster under Spin than the parametrized model itself.

We would like to be able to check larger configurations with more threads running longer programs, although experience shows that small configurations are often sufficient, since bugs found with large configurations can usually be demonstrated on small configurations after sufficient thought. Recent advances in parametrized verification [4], [22], [37] might allow us to reduce the problem of checking large configurations with n threads to checking small configurations like those we can check now. A recent theoretical result [10] reduces the problem of checking a deferred-update implementation of transactional memory with n threads to checking just two threads accessing two variables. Unfortunately, that result does not apply to our work: McRT uses update-in-place which is harder to validate than deferred-update, since updates by an aborting transaction can be visible to other transactions if the aborting transaction does not clean up properly. Coming work [11] describes an extension of this reduction to update-in-place and applies it to an abstract model of McRT.

We would also like to check programs with non-transactional loads and stores, like the code in Figure 1 illustrating the privatization problem. We note that most implementations do not allow loads and stores to occur outside of a transaction, since if a rogue thread can scribble all over memory outside a transaction and outside the control of a transactional memory implementation, then what correctness condition could possibly hold? The McRT STM architects have an idea, however, and we would like to extend our work to check their notion of correctness.

The remainder of the paper is organized as follows. We begin with a survey of the related work the Section 2, then present the McRT STM implementation in Section 3 and our Spin model in Section 4. We describe our experience with Spin and our Spin preprocessor `spp` in Section 5 along with some ideas for extending our work, and end with some concluding remarks in Section 6.

2. Related work

Transactional memory (TM) and software transactional memory (STM) were introduced by Herlihy and Moss [17] and Shavit and Touitou [36]. A number of high-performance STM implementations have been developed [34], [7], [16], [9], leading researchers to propose adding transactions to languages as programming language constructs [13], [1], [15], [38]. One approach has been to integrate STM into a language with a powerful type system, and to use type inference supported by the type system to reason about the correctness of transactional programs [14], [32]. McRT STM targets languages like C and Java without such powerful type systems, and we believe this is the first paper to address the correctness of STM implementations for such languages.

Despite the popularity of transactional memory, there is surprisingly little prior work on formal verification of transactional memory implementations. Alur *et al* [2] established some important theoretical results on the complexity of checking serializability, linearizability, and sequential consistency. Cohen *et al* [6] gave a practical method to verify the correctness of transactional memory implementations using the TLA+ [23] model checker, and verified several implementations from the literature. Their correctness conditions addressed not just serializability but also specific notions of conflicts as characterized by Scott [35]. Their work models transactional memory quite abstractly, however, whereas we try to model McRT STM at the same level of abstraction as the implementation itself to find as many bugs as possible. Other interesting approaches to verifying shared memory correctness include the following. At Sun [8], researchers

have used the PVS theorem prover to verify the correctness of a lock-free queue implementation by Michael and Scott [31]. At Princeton and Intel [3], researchers have proposed a run time validation methodology for ensuring the end-to-end correctness of transactional memory implementations. At Sun and Stanford [30], researchers have used an axiomatic formulation of a memory model with randomized testing to find bugs in a transactional memory implementation. Finally, work at EPFL gives a correctness condition intended to rule out bugs like the privatization problem given above [12], and gives a set of conditions under which the verification problem reduces to two threads and two variables [10], [11].

3. McRT STM implementation

The McRT STM implementation came to us in the form of detailed C++ pseudocode along with easy access to the implementers (see Figure 2 for a representative pseudocode fragment). We modeled the pseudocode as exactly as we could, even down to modeling pointer dereferencing with array indexing. As usual with memory protocols like cache coherence protocols, we modeled only reads and writes to single blocks of memory (ignoring unaligned accesses and multi-block accesses), and modeled only reading and writing the whole block (ignoring partial accesses such as two separate writes to the upper and lower halves of a memory block). We assumed a simple conflict manager that arbitrarily chooses one of two conflicting transactions to abort.

The fundamental idea in this protocol is to use timestamps to detect conflicts. Timestamps are everywhere:

1) *Global timestamp*: There is a single global timestamp that we denote `global.ts` that advances whenever a transaction tries to commit or abort. When this timestamp changes, it is a hint that memory may have changed, so transactions should proceed with caution.
2) *Local timestamp*: Each transaction has a local timestamp that we denote `my.ts` when the identity `my` of the transaction is understood. It records the transaction start time (the global timestamp at transaction start). It is read by other transactions when they commit, so it is global data, and it is stored in a *transaction descriptor* along with other information like the transaction read set, write set, and undo log.
3) *Memory block timestamp*: Each memory block has a timestamp that we denote `blk.ts` when the identity `blk` of the memory block is understood. It records the commit time of the last transaction writing the block. It is stored in a *transaction record* along with a lock that must be held to write the block.

In addition to timestamps, the protocol depends on two design rules.

The first design rule is that no transaction — not even aborting transactions — ever reads inconsistent data (similar to the correctness condition *opacity* proposed by Guerraoui and Kapalka [12]). This requires frequent checks of the transaction read set to validate that the read set is still valid. This validation is performed by a procedure

```
validate() = {
  ts := global.ts
  for each blk in my.readSet {
    confirm blk is locked only by me
    confirm blk.ts <= my.ts
    abort if either confirmation fails
  }
  my.ts := ts
}
```

This validation confirms that the read set has not changed since transaction start, since the local timestamp `my.ts` records the transaction start time and the memory block timestamp `blk.ts` records the commit time of the last transaction to write the block `blk`. In fact, since the read set has not changed, the transaction would generate the same result if it started now, so validation updates the local timestamp `my.ts` and pretends as if it did.

The second design rule is that no transaction commit completes until all conflicting transactions have had a chance to finish aborting. This allows conflicting transactions to undo any conflicting changes made by the transactions, and therefore avoids the linked list privatization bug illustrated in the introduction. The commit code calls a quiescence procedure

```
quiesce(ts) = {
  for each active transaction txn
    block while txn.ts < ts & txn active
}
```

to wait for transactions conflicting with `ts` to terminate. Each conflicting transaction will validate during commit, this validation will fail and abort the transaction, and the transaction will undo its changes during abort.

Commit is simple for a read-only transaction: it just sets its local timestamp to 0 to indicate to other transactions that it is no longer active. A writing transaction does a fetch-and-increment on the global timestamp, validates its read set if the global timestamp is larger than its local timestamp (indicating that memory may have changed since the start of the transaction), and sets the timestamp of every block in its write set to the global timestamp. The validation step may fail, in which case validation causes the transaction to abort and restart. If not, the transaction sets its local timestamp to 0 and waits for quiescence.

Abort is straightforward. An aborting transaction undoes changes to memory, and sets the timestamp of every block in its write set to the global timestamp. It advances the global

timestamp and sets its local timestamp to 0.

Both read and write have a fast path and a slow path, and fall back to the slow path at any hint of conflict.

The read fast path just reads the memory block, adds the block to the read set, and returns the value of the memory block, unless one of two things happens: If the block is locked for writing by another transaction or if the timestamp for the block is higher than the transaction's local timestamp (indicating the block might have changed since the transaction started), the read falls back to the slow path. The slow path just loops reading the memory block until a consistent value can be read, but since we assume the conflict handler aborts if an inconsistent value is found, this loop terminates quickly.

A write must acquire the write lock for the block, record the current value of the block in the undo log, and write the new value to the block. Acquiring the lock can follow a fast path or a slow path. The fast path falls back to the slow path if the lock is held by another transaction, if the block timestamp is greater than the transaction's local timestamp (indicating that the block may have changed since the start of the transaction), or a compare-and-swap fails to acquire the write lock for the block. In this case, because we assume the conflict handler aborts in the presence of conflict, the slow path reduces to aborting if the lock continues to be held by another transaction, if validation of the read set fails, or if a second attempt to acquire the lock with a compare-and-swap fails.

4. McRT STM model

Our model of McRT STM consists of n *program* threads, n *McRT* threads, and *shared memory*. Our model is an invocation/response model in which a program thread sends an invocation to a corresponding McRT thread, and waits for a response. It is the McRT thread that maintains transaction state (eg, the transaction's local timestamp) and reads and writes shared memory on behalf of the program thread. A program thread sends a read or write invocation to its McRT thread, and the McRT thread responds with a read or write response, or perhaps with an abort response. A program thread begins and ends a transaction by sending a start or commit invocation, and the McRT thread responds with a start or commit response, or perhaps with an abort response.

In Spin, we represent each McRT thread as a separate Spin process, but we encapsulate all of the program threads into a single, nondeterministic environment process that engages in the invocation/response interaction with the McRT threads on behalf of the program threads. Our model of the environment has several properties that are crucial to the performance we achieve with Spin.

First, the environment generates on-the-fly the programs being run by the program threads. If the ith instruction in the program is a read, the environment sends a read invocation to the McRT thread, and records the response when it arrives. At this point, the environment nondeterministically chooses an instruction for the $i+1$st instruction, if it has not already been determined, and continues. One of the early performance mistakes we made was generating complete programs before starting to run them.

Second, the environment simulates execution of each program thread, as just described, including aborts. If the environment receives an abort in response to a request on behalf of program thread i, the environment resets the thread's program counter to the start of the transaction, erases the history of results received so far, and restarts the transaction with a start invocation. We note that McRT STM actually implements an abort using pairs of setjmp/longjmp instructions to reset the program state, and this is how we model the semantics of a C++ longjmp instruction. One of the early performance mistakes we made was to regenerate the transaction from scratch after an abort, rather than simply restarting the already partially-generated transaction.

Finally, when all program threads have terminated, the environment confirms serializability of the execution by finding a transaction ordering consistent with the transaction results and shared memory. Finding the ordering is easy, because we instrument the commit response from a McRT thread with an *ordering hint* that allows the McRT thread to use state information (eg, a timestamp) to indicate where in the total ordering the committing transaction should appear. Verification that this total ordering is a serialization can be done in a single state transition in Spin.

For the McRT threads themselves, we have said several times that we modeled the McRT STM pseudocode "exactly," at least as exactly as possible, even down to modeling pointer dereferencing. The *least* exact match between the pseudocode and our model is the abort procedure illustrated in Figure 2, but even here it is easy do block-by-block pattern matching between the pseudocode and our model and see that our model quite close to the code. Remember that the transaction's local timestamp is stored in a data structure called the "transaction descriptor" that is identified by a pointer to the structure in memory. In our model, the pointer `txnDescPtr` is dereferenced by the expression `txnDesc(txnDescPtr)` which has the effect of mapping the pointer to an element of an array of transaction descriptors. The most dramatic difference between the pseudocode and Spin code is in the longjmp semantics: Where the pseudocode, after a suitable back-off process, does a longjmp to reset the state of the program thread, we reinitialize the transaction's transaction descriptor and let the environment model the resetting of program state.

5. McRT STM correctness

Our main result is the following theorem:

```
STMTxnAbort(TxnDesc* txnDesc, uint32 reason) {

  for ( (addr, val, size) in txnDesc->undoLog ) {
    if (addr is on dead stack frames) continue;
    switch(size) {
       case 4: *(uint32*)addr = val; break;
       ...
    }
  }

  if ((token  = txnDesc->token) == 0)
    token = lockedIncrement(globalTimeStamp);

  for ( txnRecPtr in txnDesc->writeSet )
    *txnRecPtr = token;









  txnDesc->localTimeStamp = 0;

  backoff();
  abortInternal(txnDesc); /* longjmp */
}
```

```
inline abortTransaction(txnDescPtr, ...) {

  foreach adr in 0..(num_addresses)-1 {
    if
      :: txnDesc(txnDescPtr).undoLog[adr] != null_data ->
           memory[adr] = txnDesc(txnDescPtr).undoLog[adr];
      :: else
    fi
  };

  fetch_and_incr (globalTimeStamp,token,token_new);


  foreach blk in 0..(num_memory_blocks)-1 {
    if
      :: txnDesc(txnDescPtr).writeSet[blk] ->
           txnRecHeap[blk] = token_new;
      :: else
    fi
  };

  /* reset transaction descriptor for restart */
  initTxnDesc(txnDesc(txnDescPtr),...);

  txnDesc(txnDescPtr).localTimeStamp = 0;


}
```

Figure 2. The abort procedure described in detailed pseudocode by the protocol architects on the left and in Spin on the right, formatted to simplify block-by-block comparison of the two representations.

Theorem 1: McRT STM guarantees the serializability of every execution of every purely-transactional program consisting of two threads each running one transaction consisting of at most three reads or writes. (By purely-transactional, we mean that every read and write occurs within a transaction.)

Remark 2: Spin validates Theorem 1 using 1:06 hours and 3.8G memory on an IBM IntelliStation Z Pro workstation running SUSE Linux 2.6.5 with a 2.6Ghz Intel Xeon processor.

Achieving the performance described in Remark 2 required many attempts at modeling shared memory protocols in Spin. It took several approaches to protocol modeling before we could model check programs of length two without blowing through eight gigabytes of memory, let alone programs of length three. To see just how successful our model of the environment is, note that there are over 10,000 programs consisting of two transactions of length three. Spin takes 45 seconds to model check all executions of a single program, yet with our model of the environment, Spin can model check all executions of all programs in just over an hour.

Specifying the notion of serializability used in the statement of Theorem 1 (the order in which transactions should be serialized) required nontrivial thought and several tries to get right. We began trying to prove *strict serializability*, as we had just proven that Dynamic STM [16] by Herlihy *et al* was strictly serializable as a warm-up exercise, but Spin demonstrated that McRT STM is not strictly serializable. We tried following the advice of the architects to use *validation order*, meaning to order transactions by the timestamp of the last successful validation to the read set, but this did not quite work. Finally, we chose *block timestamp order* which orders transactions by the highest timestamp ready or written by a transaction (remember that each block of memory has a timestamp). With the added twist that if two transactions see the same maximum timestamp, the writing transaction is ordered first, this worked. In the future, in addition to serializability, we would like to specify and validate the protocol design rules, such as the rule that no transaction — even an aborting transaction — sees inconsistent data.

There are many extensions that would be interesting to explore. One extension would be to allow reads and writes to appear outside of a transaction. What correctness condition could possibly hold when reads and writes are no longer under the control of the transactional memory implementation? This is interesting work, primarily because it requires formalizing the potentially subtle correctness conditions the protocol architects have in mind.

Another extension would be to model check larger configurations with more threads running longer programs. Performance is the primary obstacle to checking larger configurations. This leads to the question of why we chose Spin as our model checker, what problems we encountered, and how we addressed them.

5.1. Spin

Explicit state model checkers still seem to be the most effective tools for model checking protocols. There are many symbolic approaches to model checking that have been

successfully applied to hardware (see [5] for a survey), but experience shows that symbolic representations like binary decision diagrams (BDDs) that behave well on hardware models tend to blow up on protocol models (especially the highly nondeterministic protocols that appear in distributed computing).

Explicit state model checkers, however, are known to suffer from a host of problems ranging from storing the reachable state space (often solved with state hashing) to redundant interleavings (often solved with partial order reduction). While there are many experimental implementations of these optimizations, there is no single public domain model checker that implements all or even a healthy subset of them.

The big names in public-domain explicit-state model checking are currently TLA+ [23] and the related +CAL [24], [25], Murphi [21], Spin [18], and I/O Automata [29], [27] and the associated IOA toolkit [28]. TLA+ and +CAL have been used effectively in the industry when the protocol is small enough and modeled abstractly enough to fit within the model checker's capacity, but performance is an issue. Murphi and IOA also have a strong track record, especially with message-passing protocols, but the guarded-command style they use for modeling protocols is cumbersome for modeling software and shared-memory protocols. Atomic actions in message-passing protocols are relatively independent events typically triggered by message receipt. Atomic actions in shared memory protocols are typically sequentially ordered events involving a read or write of shared memory, and multi-threaded software is typically described in terms of sequential code making nested method invocations. While all of this can be encoded in a guarded-command style [25] by explicitly modeling program counters, loop index variables, run time execution stacks, etc, doing so by hand for any reasonably-sized program is cumbersome and error-prone.

Spin has a simple procedural language making it easy to model sequences of atomic actions, and, in particular, shared-memory software. It is a highly-engineered tool most famous for its implementation of bit-state hashing (reducing the space needed to store the reachable states) and partial order reduction (reducing the number of interleavings of atomic actions that must be examined). For these reasons, we were led to try Spin on this application, but ran into several problems.

5.2. Timestamps

The first problem is the fact that McRT STM uses timestamps everywhere. Timestamps are known to be the kiss of death for model checking since they are essentially unbounded counters that cause the state space to explode. In the case of McRT STM, consider a transaction that starts but is immediately aborted by the system before reading or writing any memory location. Aborting the transaction has the effect of incrementing the global timestamp but leaving all other timestamps unchanged. The two states (before starting and after aborting the transaction) differ only in the global timestamp, and the global timestamp is larger than all other timestamps in both states. Since the only operation on timestamps is comparison for order, the two states are essentially equivalent states and should be identified. We have implemented a timestamp abstraction in Spin that we hope will ameliorate this problem. We would like to make this abstraction a part of Spin itself, but extending the highly-optimized Spin code base does not look easy.

5.3. Interleavings

The second problem is a very steep performance curve that remains even after months of modeling. With our current model, checking a configuration of 2 threads running programs of length 2 takes less than a minute (22 seconds), 2 threads running programs of length 3 takes just over an hour (1:06), and 2 threads running programs of length 4 burns through eight gigabytes of memory in four hours without terminating.

The problem is clearly the length of the executions (450 transitions long for 2 threads running programs of length 3) and the number of executions resulting from the interleaving semantics. We made some mistakes that contribute to this problem. For example, it was a mistake to have explicit start and commit invocations in our invocation/response model of the environment, and we should instead have piggy-backed these invocations on the first and last memory operations in the transaction.

But Spin itself is famous for dealing with both of these problems using techniques called statement merging and partial order reduction. The problem is that Spin's implementation is so conservative that the use of iteration and global variables in the model inhibits statement merging and partial order reduction, respectively. Unfortunately, parameterized models depend on iteration, and shared memory is nothing but a large collection of global variables, so we do not benefit from Spin's highly-engineered optimizations.

5.4. Spin preprocessor

To overcome these problems, we wrote a Spin preprocessor called `spp` that mechanically transforms our fully-parameterized model into instantiated models that Spin can run more efficiently.

First consider statement merging. Statement merging makes use of the observation that a linear sequence of state transitions that access only local variables can be replaced with a single state transition. Our problem was that in Spin's conservative implementation, statement merging does not

extend over alternation or looping constructs [19], meaning that given two equivalent code sequences

```
memory[0] = 0; memory[1] = 0;
memory[2] = 0; memory[3] = 0;
```

and

```
adr = 0;
do
  :: adr < num_adrs -> memory[adr] = 0
  :: else -> break
od;
```

with `num_adrs` is set to 4, statement merging will apply to the first sequence but not the second. Unfortunately, when writing a general, parametrized model, the second code sequence is what you want to write.

We introduced a construct `foreach var in vals {body}` which we can rewrite with `spp` as straight-line code consisting of, for each value v in the list of values `vals`, a copy of `body` with the variable `var` textually replaced by the value v. The list of value `vals` can be given as a range `expr1..expr2` or list `expr1,expr2,...` where the expressions must evaluate to constants via elementary arithmetic and Boolean operations. Simply adding this construct to the language improved the performance of our already highly-optimized model by 40-50% depending on the configuration.

This construct makes it easier to write parametrized models in Spin. With this goal in mind, the complete list of constructs added are

- `foreach var in vals {body}`: execute `body` with `var` set to each value in `vals`.
- `forsome var in vals {body}`: execute `body` with `var` set to some value in `vals`.
- `forall var in vals (expr)`: the conjunction of `expr` with `var` taking on values in `vals`.
- `exists var in vals (expr)`: the disjunction of `expr` with `var` taking on values in `vals`.

Now consider partial order reduction. The problem is that the use of global variables inhibits partial order reduction [20], and shared memory is the ultimate global variable. Yet if we examine traces of our model, it is easy to see that individual processes are simply performing blocks or sequences of steps of the form `global,local,local,...` where the first step accesses a global variable and the remaining only local variables. It seems possible to rewrite our model so that each process is simply a sequence of such blocks, and that Spin can execute each such block as a single atomic transition. We hope to implement this in the near future.

6. Conclusion

We have demonstrated that model checking an industrial transactional memory implementation is possible at a level of abstraction much closer to actual implementation than normally occurs in the protocol verification literature. We have achieved the first milestone — model checking small configurations of purely transactional programs — in what promises to be an exciting sequence of milestones, including specifying and verifying the correctness of programs with nontransactional reads and writes. We have described a Spin preprocessor `spp` that improves the performance of our Spin model, and makes it possible (or at least easier) to write a general, parametrized model of a protocol that performs well under Spin. We are pleased with our experience with Spin, but we are eager to try other model checking technologies like SMT-based bounded model checking as they become available.

Acknowledgments: We thank three anonymous referees for many helpful questions and comments.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. S. Menon, B. R. Murphy, B. Saha, and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2006, pp. 26 – 37.

[2] R. Alur, K. L. McMillan, and D. Peled, "Model-checking of correctness conditions for concurrent objects," *Information and Computation*, vol. 160, no. 1-2, pp. 167–188, 2000, a preliminary version appeared in LICS'96.

[3] K. Chen, S. Malik, and P. Patra, "Runtime validation of transactional memory systems," in *International Symposium on Quality Electronic Design*, Mar. 2008.

[4] C.-T. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *Fourth International Symposium on Formal Methods in Computer-Aided Design (FMCAD)*, ser. LNCS, vol. 3312. Springer, 2004, pp. 382–398.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

[6] A. Cohen, J. O'Leary, A. Pnueli, M. R. Tuttle, and L. Zuck, "Verifying correctness of transactional memories," in *Seventh International Symposium on Formal Methods in Computer-Aided Design (FMCAD)*, M. Sheeran and J. Baumgartner, Eds., Nov. 2007, pp. 37–44.

[7] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, Sep. 2006, pp. 194–208.

[8] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Formal verification of a practical lock-free queue algorithm," in *Proceedings of the 24th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, Sep. 2004, pp. 97–114.

[9] R. Ennals, "Software transactional memory should not be obstruction-free," 2005, http://www.cambridge.intel-research.net/ rennals/notlockfree.pdf. [Online]. Available: http://www.cambridge.intel-research.net/ rennals/notlockfree.pdf"

[10] R. Guerraoui, T. A. Henzinger, B. Jobstmann, and V. Singh, "Model checking transactional memories," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2008.

[11] R. Guerraoui, T. A. Henzinger, and V. Singh, "Software transactional memory on relaxed memory models," in *Proceedings of the 21th International Conference on Computer Aided Verification (CAV)*, Jul. 2009.

[12] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Feb. 2008, pp. 175–184.

[13] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2003, pp. 388–402.

[14] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Jun. 2005, pp. 48–60.

[15] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi, "Optimizing memory transactions," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2006, pp. 14 – 25.

[16] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing (PODC)*, Jul. 2003, pp. 92–101.

[17] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the Twentieth Annual International Symposium on Computer Architecture (ISCA)*, May 1993, pp. 289–300.

[18] G. Holzman, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[19] G. J. Holzmann, "The engineering of a model checker: the gnu i-protocol case study revisited," in *Proceedings of the 6th Spin Workshop*, Sep. 1999.

[20] G. J. Holzmann and D. Peled, "An improvement in formal verification," in *Proceedings of the International Conference on Formal Description Techniques (FORTE)*, 1994, pp. 197–211.

[21] C. N. Ip and D. L. Dill, "Better verification through symmetry," in *Proc. Conf. on Computer Hardware Description Languages and their Applications*, 1993, pp. 97–111.

[22] S. Krstic, "Parameterized system verification with guard strengthening and parameter abstraction," in *Automated Verification of Infinite State Systems*, 2005.

[23] L. Lamport, *Specifying Systems*. Addison-Wesley, 2002.

[24] ——, "Checking a multithreaded algorithm with +CAL," in *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, Sep. 2006, pp. 151–163.

[25] ——, "The +CAL algorithm language," Feb. 2008, unpublished manuscript available at http://research.microsoft.com/users/lamport/pubs/pluscal.pdf.

[26] J. R. Larus and R. Rajwar, *Transactional Memory*. Morgan & Claypool, 2006.

[27] N. Lynch, *Distributed Algorithms*. Morgan Kauffman, 1996.

[28] N. Lynch *et al.*, "IOA language and toolset," tools and documentation vailable at http://groups.csail.mit.edu/tds/ioa.

[29] N. A. Lynch and M. R. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1987, pp. 137–151.

[30] C. Manovit, S. Hangal, H. Chafi, A. McDonald, C. Kozyrakis, and K. Olukotun, "Testing implementations of transactional memory," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2006, pp. 134–143.

[31] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 51, no. 1, pp. 1–26, 1998.

[32] K. F. Moore and D. Grossman, "High-level small-step operational semantics for transactions," in *The Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[33] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, V. Menon, T. Shpeisman, M. Rajagopalan, A. Ghuloum, E. Sprangle, A. Rohillah, and D. Carmean, "Runtime environment for terascale platforms," *Intel Technology Journal*, vol. 11, no. 3, pp. 207–215, Aug. 2007.

[34] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrt-stm: a high performance software transactional memory system for a multi-core runtime," in *Proceedings of the Eleventh Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Mar. 2006, pp. 187–197.

[35] M. Scott, "Sequential specification of transactional memory semantics," in *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[36] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the Fourteenth Annual Symposium on Principles of Distributed Computing (PODC)*, Aug. 1995, pp. 204–213.

[37] M. Talupur and M. Tuttle, "Going with the flow: Parameterized verification using message flows," in *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Nov. 2008, pp. 69–76.

[38] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai, "Code generation and optimization for transactional memory constructs in an unmanaged language," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Mar. 2007, pp. 34–48.