

Model Checking Transactional Memories *

Rachid Guerraoui Thomas A. Henzinger Barbara Jobstmann Vasu Singh

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Model checking software transactional memories (STMs) is difficult because of the unbounded number, length, and delay of concurrent transactions and the unbounded size of the memory. We show that, under certain conditions, the verification problem can be reduced to a finite-state problem, and we illustrate the use of the method by proving the correctness of several STMs, including two-phase locking, DSTM, TL2, and optimistic concurrency control. The safety properties we consider include strict serializability and opacity; the liveness properties include obstruction freedom, livelock freedom, and wait freedom.

1. Introduction

Transactional memory was first introduced as a hardware design by Herlihy and Moss. Later Shavit and Touitou introduced software transactional memory (STM), a software-based variant of the concept which enables a new concurrent programming model. The model aims at providing a sequentiality illusion to the programmer while giving maximal flexibility to the compiler.

Precisely because STM algorithms encapsulate the difficulty of handling concurrency, the potential of subtle errors is enormous. This makes STM a ripe and important proving ground for formal verification. We believe that an approach to formalizing and verifying STM algorithms can only have impact if it is accepted by the transactional memory community, and this concern has guided our decisions in choosing the model. We present a summary of our work [GHJS08] on verifying STM safety and liveness properties.

First, we formalize the correctness requirements of STM algorithms. For safety, we consider *strict serializability* and *opacity* [GK08]. Strict serializability preserves the order of conflicting operations between transactions, and the order of non-overlapping transactions. Opacity ensures, in addition, that aborting transactions do not see an inconsistent state of the memory, which can be disastrous in STMs (due to infinite loops, or exceptions). It should be noted that previous formalizations [Sco06] use stronger notions of safety. We look at opacity as it seems to correspond closest to the emerging consensus in the transactional software community [GK08].

Secondly, we exploit the structural symmetries that are inherent in STM algorithms to reduce the verification of unbounded STM state spaces to a problem that involves only a small number of threads and shared variables. We show that every STM that enjoys certain structural properties either violates any of the considered safety and liveness requirements on some program with two threads and two shared variables, or satisfies the requirement on all programs. The structural properties are fulfilled by the STM algorithms we have looked at: two-phase locking, DSTM [HLMS03], TL2 [DSS06], and optimistic concurrency control [KR81].

Thirdly, we define two finite-state transition systems that generate exactly the strictly serializable (resp. opaque) executions of pro-

grams with two threads and two shared variables. These transition systems can be viewed as most liberal *reference STM algorithms* guaranteeing strict serializability (resp. opacity). We show that an STM algorithm is strictly serializable (resp. opaque) iff for a specific, most general program with two threads and two variables, all executions are permitted by the reference STM algorithm. To check language containment between a given STM algorithm and the reference algorithm, we check for the existence of a simulation relation between both transition systems. The existence of a simulation relation is a commonly used, efficient sufficient condition for language containment. We implemented a simulation checker that automatically verifies strict serializability for optimistic concurrency control and opacity for two-phase locking, DSTM, and TL2 in less than 30 minutes. We observe that correctness is not self-evident in many STM algorithms, as we demonstrate later in our experiments.

The liveness requirements we consider are the standard notions of *obstruction freedom*, *livelock freedom*, and *wait freedom*. Again, we prove a structural reduction theorem to reduce the problem to checking liveness properties on the most general program with two threads and one variable, which is done by a model checker.

Related Work. There has been recent independent work on the formal verification of STM algorithms [COP⁺07]. Cohen et al. model checked STMs applied to programs with a small number of threads and variables against the strong safety criteria of Scott [Sco06]. They do not offer a reduction theorem and do not consider liveness properties.

2. Framework

Transactions. We consider a set of variables and a set of threads. The threads can issue commands to read or write a variable, or commit or abort a transaction. We define a statement to be a command executed by a particular thread, and define a word as a finite sequence of statements. The projection of a word upon the statements of a thread can be viewed as a sequence of transactions. In a given word, a transaction may be committed, aborted, or unfinished. We define the committing part of a word as the projection on all committed transactions. A statement s_1 of transaction x and a statement s_2 of transaction y (where x is different from y) *conflict* in a word w if (i) s_1 is a global read of some variable v (i.e., v has not been written before in this transaction), and s_2 is a commit, and y writes to v , or (ii) s_1 and s_2 are both commits, and x and y write to the same variable v . Two words are strictly equivalent if (i) the order of commands within threads and the order of conflicting statements of transactions is preserved, and (ii) the order of non-overlapping transactions is not reversed. A word is *strictly serializable* if its committing part is strictly equivalent to some sequential word. Furthermore, a word is opaque if it is strictly equivalent to some sequential word. An infinite word is strictly serializable (resp. opaque) if all its finite prefixes are strictly serializable (resp. opaque). We note that if a word is opaque, then it is strictly serializable.

Transactional memories. We consider thread programs as our basic sequential unit of computations. We define thread programs as

* This research was supported by the Swiss National Science Foundation. This work is published in PLDI 2008.

infinite binary trees labeled with commands. This makes the representation independent of the specific control flow statements, such as exceptions for handling aborts of transactions. For every command of a thread, we define two successor commands, one if the command is successfully executed, and another if the command fails due to an abort of the transaction. We use a set of thread programs to define a multi-threaded program. A *transactional memory (TM)* is a function from the set of programs to the set of infinite words. A transactional memory *ensures strict serializability (resp. opacity)* for all programs with n threads and k variables if for every program on n threads and k variables, every word produced by the TM is strictly serializable (resp. opaque). Moreover, a transactional memory *ensures strict serializability (resp. opacity)* if it ensures strict serializability (resp. opacity) for all programs with an arbitrary number of threads and variables.

TM algorithms. We use state transition systems to define TM. A TM algorithm is a family of TM transition systems, one for n threads and k variables, for every n and k . A TM transition system consists of a set of states, an initial state, an extended set of commands depending on the underlying TM, a pending function, and a transition relation between the states. The extended commands include the set C of commands, and TM specific additional commands. For example, a given TM may require that a thread locks a variable before writing to the variable, or that a thread validates the variables read in a transaction, before accessing a new variable. Every extended command is assumed to execute atomically.

A TM algorithm interacts with a program and a scheduler. The scheduler chooses a thread, which determines the next command to be executed. The TM transition system decides whether the command can be executed in a single atomic step, or in several atomic steps (using additional extended commands), or has to be aborted. The TM algorithm gives back to the program a response. The response tells whether the TM algorithm needs additional steps to complete the command, or whether the TM algorithm needs to abort the transaction, or whether the TM algorithm has completed the command. Given a program, a scheduler, and a TM transition system, we get a run. Projecting the run to the set of successful statements (that is, aborts, and statements that get response 1) gives an infinite word. We describe the language of a TM transition system as the set of infinite words that it can produce for any program and any scheduler.

A TM algorithm defines a transactional memory such that for all n and k , for every program on n threads and k variables, the TM produces a word iff the word corresponds to some run of the TM algorithm for the given program. It follows that a TM ensures strict serializability (resp. opacity) for all programs with n threads and k variables iff all words in the language of the corresponding TM algorithm are strictly serializable (resp. opaque).

3. Structural properties

We briefly describe some structural properties that are satisfied by all TMs we have studied, which establish the subsequent theorem.

- Aborting and unfinished transactions can influence other transactions only by forcing them to abort.
- For non-overlapping transactions, the TM is oblivious to the identity of the thread executing the transaction.
- If a transaction can commit, then removing all statements that involve some particular variables does not cause the transaction to abort.
- If a word is allowed by the TM, then more sequential forms of the word are also allowed.

Table 1. Time for simulation checking for TM algorithms on a quad dual core 2.8 GHz server with 16 GB RAM. In case simulation holds, we write (YES, time to find the simulation). Otherwise, we write (NO, counterexample produced, time to prove there is no simulation, time to find the counterexample). A ‘*’ for the search for simulation relation means that it does not complete in 2 hours. The RSS and RO TM transition systems have 12346 and 9202 states respectively.

TM transition system	Number of states	Simulated by RSS	Simulated by RO
seq	3	YES, 0.8s	YES, 0.7s
2PL	99	YES, 13s	YES, 8s
DSTM	944	YES, 127s	YES, 82s
TL2	11840	YES, 1647s	YES, 1438s
OCC	4480	YES, 765s	NO, w_1 , 567s, 4s
TL2 modified	17520	NO, w_2 , *, 9s	NO, w_2 , *, 8s

Counterexamples: $w_1 = (w, 1)_2, (r, 1)_1, c_2, (r, 1)_1$
 $w_2 = (w, 2)_1, (w, 1)_2, (r, 2)_2, (r, 1)_1, c_2, c_1$

Theorem 1. If a TM ensures strict serializability (resp. opacity) for all programs on two threads and two variables, and satisfies the above properties, then it ensures strict serializability (resp. opacity).

4. The reference TM algorithms

The key to our model checking procedure are the *finite state* reference TM algorithms for strict serializability and opacity for two threads and two variables. The construction is non-trivial, given that threads may be delayed arbitrarily, transactions may contain arbitrarily many statements and may be aborted arbitrarily often. To get around the problem of infinite states, we maintain *prohibited read and write sets* for every thread. These sets allow to handle unbounded delay between transactions, as committing transactions store the required information in the sets of other threads. The reference strictly serializable (RSS) TM transition system is based on the following observation: *Every committing transaction serializes at some point during its execution.* The RSS TM transition system makes a non-deterministic guess of when a transaction serializes. Depending upon the guess, the transition system checks upon the commit of a transaction, whether the commit can be executed, or it needs to abort.

Apart from the requirements of the above mentioned reference TM algorithm for strict serializability, opacity requires that even global reads of aborting transactions observe consistent values. It turns out that we can obtain a finite-state representation of the RO TM transition system by slightly modifying our RSS TM transition system. The RO TM transition system is based on the following observation: *Every committing and aborting transaction should serialize at some point during its execution.* Like the RSS TM transition system, the RO TM transition system makes a non-deterministic guess of when a transaction serializes. In this case, the transition system checks upon every global read and every commit of a transaction, whether the command can be executed or the transaction needs to be aborted.

A TM defined by a TM algorithm ensures strict serializability (resp. opacity) iff the language of the TM algorithm for 2 threads and 2 variables is a subset of the language of the RSS (resp. RO) TM algorithm for 2 threads and 2 variables. As checking language inclusion is PSPACE-hard, we use the common technique of checking for the existence of a simulation relation between both transition systems. The existence of a simulation relation is a sufficient condition for language inclusion.

We built an automatic verification tool in C for checking the existence of simulation relations using a quadratic algorithm. The

tool is conceived as a platform for the automatic verification of TMs that satisfy the structural properties. The tool takes as input two TM algorithms and checks whether the first is simulated by the second. If the tool fails to find a simulation relation, it attempts to return a counterexample. In certain cases, it is possible that although language inclusion holds, the tool cannot find a simulation relation. Thus, our decision procedure is sound but not complete. For all TM transition systems we considered, our tool terminates after finding a simulation relation, or a counterexample. Our results shown in Table 1 establish the following result: *The sequential TM, two-phase locking, DSTM, and TL2 ensure opacity. The optimistic concurrency control ensures strict serializability, but not opacity.*

Our tool discovered a subtle point in TL2. In the description of the published TL2 algorithm, we found the order of two operations, validating the read set (*rvalidate*), and checking whether a variable in the read set is locked (*chklock*), ambiguous. We modeled these operations as two separate atomic operations, such that that *chklock* happens after *rvalidate*, to obtain a modified TL2 TM algorithm. The tool found that the modified TL2 TM algorithm is not simulated by the RSS TM transition system, and the tool provided a counterexample. Our experiments discover that the correctness of TL2 is based on the subtle fact that either the version number and the lock bit have to be accessed atomically (as in the actual TL2 implementation), or *rvalidate* has to occur after *chklock*.

5. Verifying liveness

We define two different notions of liveness, obstruction freedom and livelock freedom, as discussed in the transactional memory literature. A third notion, wait freedom, implies livelock freedom.

- An infinite word w is *obstruction free* if for all threads t , if the word w has an infinite number of aborts of t , then w has an infinite number of commits of t or there are infinitely many statements of some thread $u \neq t$.
- An infinite word w is *livelock free* if the word has an infinite number of commits, or there is a thread t such that t has infinitely many statements and finitely many aborts in w . Note that livelock freedom implies obstruction freedom.

A TM ensures *obstruction freedom* (resp. *livelock freedom*) for all programs with n threads and k variables if for every program, every word produced by the TM is obstruction free (resp. livelock free). A TM ensures *obstruction freedom* (resp. *livelock freedom*) if it ensures obstruction freedom (resp. livelock freedom) for all programs with an arbitrary number of threads and variables. We use the formalism of TM algorithms to verify liveness properties of TMs. A TM defined by a TM algorithm ensures obstruction freedom for all programs with n threads and k variables iff there is no loop l in the TM transition system for n threads and k variables such that all statements in l are from the same thread, and l contains no commit, and l contains an abort. Similarly, a TM ensures livelock freedom for all programs with n threads and k variables iff there is no loop l in the TM transition system for n threads and k variables such that l contains no commit, and every thread that has a statement in l , has an abort in l .

As we did for safety, we provide structural properties which allow us to establish the following theorem.

- A thread t running in isolation (no interleaved step from other threads) shall abort repeatedly only if it conflicts with some unfinished transaction.
- A thread t running in isolation shall abort repeatedly only if some commands corresponding to some variables are not allowed.

Table 2. Results of model checking liveness on a dual core 2.66GHz desktop PC with 2 GB RAM. The notation is similar to Table 1. The time denotes the time required to prove a liveness property or find a counterexample.

TM algorithm	Obstruction freedom	Livelock freedom
seq	NO, $w_1, 0.1s$	NO, $w_1, 0.1s$
2PL	NO, $w_1, 0.1s$	NO, $w_1, 0.1s$
DSTM	YES, $2s$	NO, $w_2, 0.2s$
TL2	NO, $w_1, 0.4s$	NO, $w_1, 0.4s$
OCC	NO, $w_3, 0.7s$	NO, $w_3, 0.7s$
Counterexamples: $w_1 = a_1, w_2 = a_1, (r, 1)_1, (o, 1)_1, a_2, (o, 1)_2$ $w_3 = s_1, a_1$		

Theorem 2. If a TM satisfies the above structural properties, and the TM ensures obstruction freedom for two threads and one variable, then it ensures obstruction freedom.

We built a verification tool to check obstruction freedom and livelock freedom properties for transaction memories defined by TM algorithms. If the liveness property fails, then the tool provides feedback in the form of a run that represents a counterexample. From Table 2, we conclude: *DSTM ensures obstruction freedom and does not ensure livelock freedom. Sequential TM, two phase-locking, TL2, and optimistic concurrency control do not ensure obstruction freedom.*

6. Conclusion

We presented a new technique for verifying STM safety and liveness properties. The cornerstones of our technique are a finite-state representation for the languages of strictly serializable and opaque executions, and an automated verification tool for STMs. Our method applies to all STM protocols that satisfy certain structural properties, and we successfully verified opacity for 2PL, DSTM, and TL2, and the obstruction freedom of DSTM.

References

- [COP⁺07] A. Cohen, J. O’Leary, A. Pnueli, M. R. Tuttle, and L. Zuck. Verifying correctness of transactional memories. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 37–44. IEEE Computer Society, 2007.
- [DSS06] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *International Symposium on Distributed Computing (DISC)*, pages 194–208. Springer, 2006.
- [GHJS08] Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In *PLDI*, 2008. to appear.
- [GK08] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 175–184, 2008.
- [HLMS03] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, 2003.
- [KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, pages 213–226, 1981.
- [Sco06] M. L. Scott. Sequential specification of transactional memory semantics. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.