Form Methods Syst Des (2011) 39:297–331
DOI 10.1007/s10703-011-0131-3

Verification of STM on relaxed memory models

Rachid Guerraoui · Thomas A. Henzinger · Vasu Singh

Published online: 23 November 2011

Abstract Software transactional memories (STM) are described in the literature with assumptions of sequentially consistent program execution and atomicity of high level operations like read, write, and abort. However, in a realistic setting, processors use relaxed memory models to optimize hardware performance. Moreover, the atomicity of operations depends on the underlying hardware. This paper presents the first approach to verify STMs under relaxed memory models with atomicity of 32 bit loads and stores, and read-modify-write operations. We describe RML, a simple language for expressing concurrent programs. We develop a semantics of RML parametrized by a relaxed memory model. We then present our tool, FOIL, which takes as input the RML description of an STM algorithm restricted to two threads and two variables, and the description of a memory model, and automatically determines the locations of fences, which if inserted, ensure the correctness of the restricted STM algorithm under the given memory model. We use FOIL to verify DSTM, TL2, and McRT STM under the memory models of sequential consistency, total store order, partial store order, and relaxed memory order for two threads and two variables. Finally, we extend the verification results for DSTM and TL2 to an arbitrary number of threads and variables by manually proving that the structural properties of STMs are satisfied at the hardware level of atomicity under the considered relaxed memory models.

Keywords Transactional memories · Model checking · Relaxed memory models

This research was supported by the Swiss National Science Foundation. This paper is an extended and revised version of our previous work on model checking transactional memories [21].

R. Guerraoui
LPD (Station 14), I&C, EPFL, 1015 Lausanne, Switzerland
e-mail: rachid.guerraoui@epfl.ch

T.A. Henzinger · V. Singh (✉)
IST Austria, Am Campus 1, Maria Gugging, Klosterneuburg, 3400, Austria
e-mail: vasu.singh@epfl.ch

T.A. Henzinger
e-mail: tah@epfl.ch

1 Introduction

Transactional memory (TM) has recently gained much interest due to the advent of multicore architectures. Inspired by how databases manage concurrency, TM was first introduced by Herlihy and Moss [23] in multi-processor design. Later Shavit and Touitou [38] introduced STM, a software-based variant of the concept. An STM allows to structure an application in terms of coarse-grained code blocks that appear to be executed atomically [23, 38]. An STM provides the illusion of sequentiality to a programmer and maximal flexibility to the underlying hardware. However, behind the apparent simplicity of the STM abstraction, lie challenging algorithms that seek to ensure transactional atomicity without restricting parallelism.

Various correctness criteria have been proposed for STM algorithms. One criterion, popular for its relevance to the STM designers, is opacity [22]. Opacity is motivated by the fact that in STMs, observing inconsistent state by even an aborted transaction can lead to unexpected side effects. Opacity builds upon strict serializability [31], a correctness property used for database transactions. Strict serializability requires that the committed transactions appear to be executed in a serial order, consistent with the order of non-overlapping transactions. Opacity further requires that even aborted transactions appear to be executed in a serial order.

Previous attempts at formally verifying the correctness of STMs [8, 19, 20] with respect to different correctness criteria assumed that high-level transactional commands like start, read, write, commit, and abort execute atomically and in a sequentially consistent manner. Verification of an STM at this level of abstraction leaves much room for errors in a realistic setting. This is because the actual hardware on which STMs run supports a finer-grained degree of atomicity: in practice, the set of atomic instructions rather corresponds to *load*, *store*, and *read-modify-write*. Furthermore, compilers and processors assume relaxed memory models [1] and are notorious for playing tricks to optimize performance, e.g., by reversing the order of instructions to different addresses. Typically, STM designers use *fences* to ensure a strict ordering of memory operations. As fences hurt performance, STM designers want to use fences only when necessary for correctness.

To illustrate some of the issues, consider the code fragments of the commit and the read procedures of a typical timestamp-based STM like TL2 [11] in Fig. 1. Assume that at the start of a transaction, t_1 and t_2 are set to the global timestamp *ts*. The commit procedure updates the timestamp *ts* before it updates the variables in the write set. The read procedure first reads the timestamp, followed by the read of the variable, followed by a second read of the timestamp. The read is successful only if the two timestamps are equal. A crucial question is, given the memory model, which fences are required to keep the STM correct. On a memory model like sequential consistency [26] or total store order [42], the code fragment in Fig. 1 is correct without fences. On the other hand, on memory models that relax store order, like partial store order [42], we need to add a *store fence* after the timestamp update in the commit procedure. For even more relaxed memory models that may swap independent loads, like relaxed memory order [42], as well as the Java memory model [29], we need more fences, namely, *load fences* in the read procedure. But the question is how many? Do we need to ensure that the read of v is between the two reads of *ts*, and thus put two fences? The answer is no. To ensure correctness, we just need one fence and guarantee that the second read of *ts* comes after the read of v.

Devising a verification technique to model check STM algorithms assuming relaxed memory models and hardware-level atomicity is challenging. A first challenge is to devise a precise and unified formalism in which the STM implementations can be expressed.

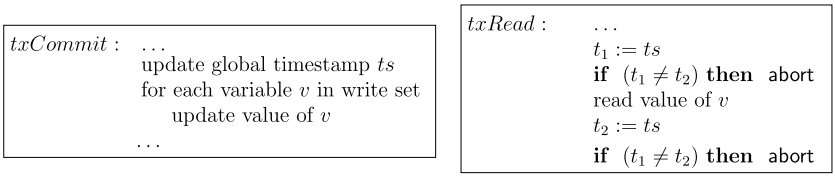

Fig. 1 Code fragments of commit and read procedures of a timestamp-based STM

A second challenge is to cope with the non-determinism explosion. Not surprisingly, when compared to verifying an STM at a high-level atomic alphabet, the level of non-determinism to be dealt with at hardware-level atomicity under a relaxed memory model is much higher. For example, the implementation of DSTM [24] with 2 threads and 2 variables generates 1,000 states with a high-level atomic alphabet [19] and 1,200,000 states with a low-level one, even on a sequentially consistent memory model. A relaxed memory model further increases the state space.

This paper takes up the challenge of bridging the gap between STM descriptions in the literature and their real implementations on actual hardware. We start by presenting a formalism to express memory models as a function of hardware memory instructions, that is, loads and stores to 32 bit words. We describe various relaxed memory models, such as total store order (TSO), partial store order (PSO), and relaxed memory order (RMO) in our formalism. The reason for choosing these memory models is to capture different levels of relaxations allowed by different multiprocessors. Unlike earlier formalisms [8, 19] used for verification, our formalism can be used to express and check the correctness of STMs with both update semantics: direct (eager) and deferred (lazy). Then, we present a new language, RML (*Relaxed Memory Language*), with a hardware-level of atomicity, whose semantics is parametrized by various relaxed memory models. Then, we describe a new tool, FOIL (a fencing weapon), to verify the opacity of three different STM algorithms, DSTM, TL2, and McRT STM, under different memory models. We choose these STMs as they represent three different and important trends in STM design. DSTM is obstruction-free (does not use locks), TL2 is a lock-based STM with deferred-update semantics, and McRT STM is a lock-based STM with direct-update semantics. While we choose opacity as the correctness criterion, using FOIL we can also verify other correctness properties such as strict serializability that can be specified in our formalism. Our verification technique consists of two parts. First, we use our automated tool FOIL to verify the correctness of STM algorithms restricted to two threads and two variables. Then, we use manual proofs of the structural properties to extend the verification results to an arbitrary number of threads and variables.

FOIL proves the opacity of the considered STM algorithms under sequential consistency and TSO. As the original STM algorithms have no fences, FOIL generates counterexamples to opacity for the STMs under further relaxed memory models (PSO and RMO), and automatically inserts fences within the RML description of the STM algorithms which are required (depending upon the memory model) to ensure opacity. We observe that FOIL inserts fences in a pragmatic manner, as all fences it inserts match those in the manually optimized official implementations of the considered STMs. Our verification leads to an interesting observation that many STMs are sensitive to the order of loads and stores, but neither to the order of a store followed by a load, nor to store buffering. Thus, while two of the STM algorithms (TL2 and McRT-STM) we consider need fences for opacity under PSO and RMO, they are indeed opaque under TSO without any fences.

At last, we extend the proof of correctness of DSTM and TL2 for two threads and two variables to an arbitrary number of threads and variables. We build upon a set of structural properties of STMs [19] and manually prove that these structural properties are satisfied for DSTM and TL2 at the hardware level of atomicity under relaxed memory models. We cannot extend the verification results for McRT STM to an arbitrary number of threads and variables as McRT STM does not satisfy the structural properties.

2 Transactional programs

We first present a general formalism to express hardware memory instructions. Then, we present a simple language RML for expressing transactional programs. We then formalize memory models which describe the interaction between memory instructions. Then, we formalize the correctness property, opacity. We define TM specifications and build one for representing opacity.

2.1 Memory instructions

Let *Addr* be a set of memory addresses. Let *Inst* be the set of *memory instructions* that are executed atomically by the hardware. We define the set *Inst* as follows, where $a \in Addr$:

$$Inst ::= \langle \mathsf{load}\ a \rangle \mid \langle \mathsf{store}\ a \rangle \mid \langle \mathsf{cas}\ a \rangle$$

We use the $\langle \mathsf{cas}\ a \rangle$ instruction as a generic read-modify-write instruction.

We introduce a high-level language, RML, to express STM algorithms with hardware-level atomicity on relaxed memory models. The key idea behind the design of RML is to have a semantics parametrized by the underlying memory model. To capture a relaxed memory model, RML defers a statement until the statement is forced to execute due to a fence, and RML reorders or eliminates deferred statements according to the memory model. We describe below the syntax and semantics of RML.

2.2 Syntax

To describe STM algorithms in RML, we use local and global integer-valued locations, which are either variables or arrays. We also have a set of array index variables. The syntax of RML is given in Fig. 2. A memory statement (denoted by *mem_stmt*) in RML models an instruction that executes atomically on the hardware. It can, for instance, be a store or a load of a global variable. Moreover, the STM specific statements are denoted by *tm_stmt*, and fence statements are denoted by *fence*. Let S_M be the set of memory statements, S_{tm} be the set of STM specific statements, and S_F be the set of fence statements in RML. Let P be the set of RML programs.

2.3 Transactional programs

Let V be a set of *transactional variables*. Let T be a set of *threads*. Let the set C of *commands* be $(\{\mathsf{read}, \mathsf{write}\} \times V) \cup \{\mathsf{xend}\}$. These commands correspond to a read or write of a transactional variable, and to a transaction end. Depending upon the underlying STM, the execution of these commands may correspond to a sequence of hardware memory instructions. For example, a read of a transactional variable may require to check the consistency

$$
\begin{array}{lll}
l & ::= & lv \mid la[idx] \\
g & ::= & gv \mid ga[idx] \\
e & ::= & f(l, \ldots, l, idx, \ldots, idx) \\
c & ::= & f(idx, \ldots, idx) \\
tm\_stmt & ::= & \mathsf{rfin} \mid \mathsf{commit} \mid \mathsf{abort} \\
mem\_stmt & ::= & g := e \mid l := g \mid l := e \mid idx := c \mid l := \mathsf{cas}(g, e, e) \\
 & & \mid \mathsf{rollback}\ g := e \\
fence & ::= & \mathsf{stfence} \mid \mathsf{ldfence} \\
p & ::= & mem\_stmt \mid tm\_stmt \mid fence \mid p\ ;\ p \\
 & & \mid \textbf{if}\ e\ \textbf{then}\ p\ \textbf{else}\ p \mid \textbf{while}\ e\ \textbf{do}\ p
\end{array}
$$

Fig. 2 The syntax of the language RML

of the variable by first reading a version number. Similarly, a transaction end may require to copy many variables from a thread-local buffer to global memory. Moreover, the semantics of the write and the xend commands depend on the underlying STM. For example, a (write, v) command does not alter the value of v in a deferred-update STM, whereas it does in a direct-update STM.

We restrict ourselves to purely transactional code, that is, every operation is part of some transaction. We consider transactional programs as our basic sequential unit of computation. We express transactional programs as infinite binary trees on commands, which makes the representation independent of 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. Note that this definition allows us to capture different retry mechanisms of STMs, e.g., retry the same transaction until it succeeds, or try another transaction after an abort. We use a set of transactional programs to define a multithreaded transactional program. A *transactional program* θ on V is an infinite binary tree $\theta : \mathbb{B}^* \to C$. A *multithreaded transactional program* $prog = \langle \theta^1 \ldots \theta^n \rangle$ on V is a tuple of transactional programs on V. Let *Progs* be the set of all multithreaded transactional programs.

2.4 STM correctness

An STM is characterized by the set of histories (sequences of memory instructions) the STM produces for a given transactional program. In order to reason about the correctness of STMs, the history must contain, apart from the sequence of memory instructions that capture the loads and stores to transactional variables in the program, the following information: (i) when transactions finish (captured with commit and abort instructions), (ii) when a read command finishes (captured with rfin instruction), and (iii) rollback of stores to transactional variables in V (captured with rollback a). The commit and abort instructions are needed to reason about the serialization of transactions. The rfin instruction is needed in formalizing opacity. For example, rfin allows to distinguish the point in time where a variable is loaded from the point where the value loaded is used. The rollback instruction is used in direct update STM to undo a store to a variable.

2.4.1 Histories

We define $\hat{Inst} = Inst_V \cup (\mathsf{rollback} \times V) \cup \{\mathsf{rfin}, \mathsf{commit}, \mathsf{abort}\}$, where $Inst_V \subseteq Inst$ is the set of memory instructions to the transactional variables V. There are two important things to

note here. The instructions {rfin, commit, abort} have no physical representation in the memory instruction sequence, but are needed to punctuate transactional reads, writes, and ends in order to check correctness. The instruction rollback is physically a store instruction, but needs to be distinguished from a store in order to check correctness. Essentially, a rollback undoes the effect of a previous store instruction.

Let $Op = \hat{Inst} \times T$ be the set of *operations*. A *history* $h \in Op^*$ is a finite sequence of operations. An STM takes as input a transactional program and, depending upon the memory model, produces a set of histories. Formally, a *software transactional memory* is a function $\Gamma : Progs \times \mathbf{M} \to 2^{Op^*}$.

A *correctness property* π is a subset of Op^*. It is natural to require that an STM is correct for *all* programs on a *specific* memory model. This is because an STM may be optimized for performance for a specific memory model, while it could be incorrect on weaker models. That is, different implementation versions may be designed for different memory models. An STM Γ is *correct* for a property π under a memory model M if for all programs $prog \in Progs$, we have $\Gamma(prog, M) \subseteq \pi$.

Given a history $h \in Op^*$, we define the *thread projection* $h|_t$ of h on thread $t \in T$ as the subsequence of h consisting of all operations op in h such that $op \in \hat{Inst} \times \{t\}$. Given a thread projection $h|_t = op_0 \ldots op_m$ of a history h on thread t, an operation op_i is *finishing in* $h|_t$ if op_i is a commit or an abort. An operation op_i is *initiating in* $h|_t$ if op_i is the first operation in $h|_t$, or the previous operation op_{i-1} is a finishing statement. Given a thread projection $h|_t$ of a history h on thread t, a consecutive subsequence $x = op_0 \ldots op_m$ of $h|_t$ is a *transaction* of thread t in h if (i) op_0 is initiating in $h|_t$, and (ii) op_m is either finishing in $h|_t$, or op_m is the last operation in $h|_t$, and (iii) no other operation in x is finishing in $h|_t$. The transaction x is *committing* in h if op_m is a commit. The transaction x is *aborting* in h if op_m is an abort. Otherwise, the transaction x is *unfinished* in h. We say that a load of a transaction variable by thread t is *used* in a history h if the load is immediately succeeded by an rfin statement in $h|_t$. Given a history h, we define $usedloads(h)$ as the longest subsequence of h such that all loads of transaction variables in $usedloads(h)$ are used. Given a history h and two transactions x and y in h (possibly of different threads), we say that x *precedes* y in h, written as $x <_h y$, if the last operation of x occurs before the first operation of y in h. A history h is *sequential* if for every pair x, y of transactions in h, either $x <_h y$ or $y <_h x$.

2.4.2 Opacity

We consider opacity [22] as the correctness (safety) requirement of transactional memories. Opacity builds upon the property of strict serializability [31], which requires that the order of conflicting operations from committing transactions is preserved, and the order of non-overlapping transactions is preserved. Opacity, in addition to strict serializability, requires that even aborting transactions do not read inconsistent values. The motivation behind the stricter requirement for aborting transactions in opacity is that in STMs, inconsistent reads may have unexpected side effects, like infinite loops, or array bound violations. Most of the STMs [11, 24, 34] in the literature are designed to satisfy opacity. However, there do exist STMs that ensure just strict serializability (for example, a variant to McRT STM), and use exception handling to deal with inconsistent reads.

For the feasibility of the verification problem, we restrict the notion of opacity with two assumptions. We describe the assumptions and justify them below. Both assumptions restrict the scope of direct update STM allowed by our formalism. However, our assumptions do not restrict the deferred update STM.

Firstly, we assume that if a store of a transactional variable rolls back some time later, then the store should not be observed by a load and should not be overwritten by another

store. In other words, we assume that a direct update STM algorithm uses exclusive locks for the variables being written. If a STM does not satisfy this assumption, we cannot verify whether the STM is correct. Moreover, a rollback instruction does not precede a store instruction, as a rollback instruction undoes the effect of a store instruction. We enforce our assumption by defining a notion of well-formedness of histories. Given a transactional variable v, we define the *variable projection* $h|_v$ of a history h on v as the longest subsequence of loads, stores, rollbacks, and compare-and-swaps to the variable v.

We say that a store or a compare-and-swap instruction *inst* to variable v in transaction x is *final* in a history h if there does not exist a rollback instruction to v after *inst* in x.

We say that a history h is *well-formed* if for all transactions x, (i) if x consists of a rollback instruction *inst* to a variable v, then x consists of a store instruction to v before *inst*, (ii) if x is an aborted transaction, then x does not consist of any final stores, and (iii) a non-final store to some variable v in x in $h|_v$ is not immediately followed by a compare-and-swap, a used load, or a store to v. This assumption is valid as all direct update STM algorithms we know of rely on locking protocols.

Secondly, we consider a prefix closed subset of opacity as our correctness notion. The justification is that all prefixes of a history produced by a STM should be correct. For example, the history $h = ((\mathsf{read}, v_1), 1), (\mathsf{rfin}, 1), ((\mathsf{read}, v_2), 2), (\mathsf{rfin}, 2), ((\mathsf{write}, v_2), 1), ((\mathsf{write}, v_1), 1)$ is not opaque by the standard definition of opacity. Now let h be suffixed by $((\mathsf{rollback}, v_2), 1)$ $((\mathsf{rollback}, v_1), 2)$ to get history h'. We note that h' is opaque, as all operations of thread 1 may precede all operations of thread 2. Note that the reason that opacity is not prefix closed are the rollback instructions. Intuitively, a rollback operation may remove conflicts from a conflict graph, and thus remove a cycle which might exist in a prefix of the history.

An operation $op_1 = (inst_1, t)$ of transaction x and an operation $op_2 = (inst_2, u)$ of transaction y (where x is different from y) *conflict* in a history h if

- $inst_1$ is a load, a final compare-and-swap, or a final store instruction to some transactional variable v and $inst_2$ is a final store instruction to v in h
- $inst_1$ and $inst_2$ are final store instructions to some transactional variable v

A history $h = op_0 \dots op_m$ is *strictly equivalent* to a history h' if (i) for every thread $t \in T$, we have $h|_t = h'|_t$, and (ii) for every pair op_i, op_j of operations in h, if op_i and op_j conflict and $i < j$, then op_i occurs before op_j in h', and (iii) for every pair x, y of transactions in h, where x is a finished transaction, if $x <_h y$, then it is not the case that $y <_{h'} x$. We define *opacity* as the set of histories h such that there exists a sequential history h', where h' is strictly equivalent to $usedloads(h)$. We specify correctness properties using transition systems called STM specifications [20].

2.5 TM specifications

A *TM specification* is a 3-tuple $\langle Q, q_{init}, \delta \rangle$, where Q is a set of states, q_{init} is the initial state, and $\delta : Q \times Op \to Q$ is a transition function. A history $op_0 \dots op_m$ is a *run* of the TM specification if there exist states $q_0 \dots q_{m+1}$ in Q such that $q_0 = q_{init}$ and for all i such that $0 \leq i \leq m$, we have $(q_i, op_i, q_{i+1}) \in \delta$. The *language* L of a TM specification is the set of all runs of the TM specification. A *TM specification Spec* defines a correctness property π if $L(Spec) = \pi$. A TM specification for opacity at a coarse-grained alphabet of read, write, commit, and abort statements was developed [20]. To verify the STM algorithms at the hardware level of atomicity, we build a new TM specification for opacity with the alphabet Op.

3 TM Specifications for opacity

Developing a TM specification at the hardware level atomicity is challenging due to the following reasons:

- A commit may consist of multiple store instructions. As soon as a transaction stores, and some other transaction loads the value or overwrites the value, the first transaction cannot abort anymore.
- There is a distinction between the point when a variable is read and when the read is declared as finished. Although a transaction reads an inconsistent value, the history may still be opaque. But, if a transaction finishes the read of an inconsistent value, the history is not opaque.
- Stores may roll back in direct update systems. For example, if a transaction has rolled back its store, then it could appear as if the store was never performed.

We use our specification to automatically prove the correctness of STM for two threads and two variables. Later, we manually extend our proofs of correctness to an arbitrary number of threads and variables using structural properties of STM. Thus, we develop the TM specification for opacity for only two threads, which keeps the TM specification simple. We first develop a nondeterministic TM specification for opacity.

3.1 A nondeterministic TM specification

We define the *nondeterministic TM specification for opacity Spec* for two threads as the tuple $\langle Q, q_{init}, \delta\rangle$. A state $q \in Q$ is a 10-tuple $\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle$, where $Status : T \to \{\mathsf{finished}, \mathsf{abortsure}, \mathsf{commitsure}\}$ is the status, $SerStatus : T \to \{true, false\}$ is the serialization status, $rs : T \to 2^V$ is the read set, $ws : T \to 2^V$ is the write set, $urs : T \to \{\bot\} \cup V$ is the unfinished read variable, $prs : T \to 2^V$ is the prohibited read set, $pws : T \to 2^V$ is the prohibited write set, $wp : T \to \{true, false\}$ is the write predecessor flag, $rp : T \to \{true, false\}$ is the read predecessor flag, and $serp : T \to \{true, false\}$ is the serialization predecessor flag for the threads. The initial state $q_{init} = \langle Status_0, SerStatus_0, rs_0, ws_0, urs_0, prs_0, pws_0, wp_0, rp_0, serp_0\rangle$, where $Status_0(t) = \mathsf{finished}$, $SerStatus_0(t) = false$, $urs_0(t) = \bot$, $wp_0(t) = rp_0(t) = serp_0(t) = false$, and $rs_0(t) = ws_0(t) = prs_0(t) = pws_0(t) = \emptyset$ for both threads. The transition relation is obtained using Algorithm 1. The thread t refers to the thread taking the step, and the thread u refers to the other thread. Given a state q, the procedure $ResetState(q, t)$ makes the following updates: (i) sets $Status(t)$ to $\mathsf{finished}$, (ii) sets $SerStatus(t)$ to $false$, (iii) sets $urs(t)$ to $\bot$, (iv) sets $rs(t)$, $ws(t)$, $prs(t)$, and $pws(t)$ to $\emptyset$, and (v) sets $rp(t)$, $wp(t)$, $rp(u)$, $wp(u)$, and $serp(u)$ to $false$.

3.1.1 Construction

We describe the rules that govern the set of runs that are produced by the nondeterministic TM specification. Let r be a run of the TM specification *Spec*. Let x be the unfinished transaction of a thread, and let y be the unfinished transaction of the other thread in the run r. The nondeterministic TM specification ensures the following:

1. A variable v is in the prohibited write set of x if there is a committed transaction z in r such that z serializes after x and z has a final store or a finished read of v
2. A variable v is in the prohibited read set of x if there is a committed transaction z in r such that z serializes after x and z has a final store of v

Algorithm 1 The nondeterministic STM specification for opacity

nondetTMSpec$(\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle, op)$

if $op = ((\mathsf{store}, v), t)$ **then**
  if $Status(t) = \mathsf{abortsure}$ **then return** $\bot$
  if $urs(t) \neq \bot$ **then return** $\bot$
  if $v \in pws(t)$ **then return** $\bot$
  $ws(t) := ws(t) \cup \{v\}$
  if $v \in ws(u)$ **then**
    if $serp(u)$ **then return** $\bot$ **else** $serp(t) := true$
    if $Status(u) = \mathsf{abortsure}$ **then return** $\bot$
    if $wp(u)$ **then return** $\bot$
    if $Status(u) = \mathsf{finished}$ **then**
      $Status(u) := \mathsf{commitsure}$
      if $urs(u) = v$ for some variable $v \in V$ **then**
        $rs(u) := rs(u) \cup \{v\}$
  if $v \in rs(u)$ **then**
    if $serp(u)$ **then return** $\bot$ **else** $serp(t) := true$
  if $v \in urs(u)$ **then**
    if $serp(u)$ **then**
      $Status(u) := \mathsf{abortsure}$; $urs(u) := \bot$
    $rp(t) := true$

if $op = ((\mathsf{load}, v), t)$ **then**
  if $Status(t) = \mathsf{abortsure}$ **then return** $\bot$
  if $urs(t) \neq \bot$ **then return** $\bot$
  if $v \in prs(t)$ **then**
    if $Status(t) = \mathsf{commitsure}$ **then return** $\bot$
    $Status(t) := \mathsf{abortsure}$
  $urs(t) := v$
  if $Status(t) = \mathsf{commitsure}$ **then** $rs(t) := rs(t) \cup \{v\}$
  if $v \in ws(u)$ **then**
    if $Status(t) = \mathsf{commitsure}$ **then**
      if $serp(u)$ **then return** $\bot$ **else** $serp(t) := true$
      if $Status(u) = \mathsf{abortsure}$ **then return** $\bot$
      if $wp(u)$ **then return** $\bot$
      if $Status(u) = \mathsf{finished}$ **then**
        $Status(u) := \mathsf{commitsure}$
        if $urs(u) = v$ for some variable $v \in V$ **then**
          $rs(u) := rs(u) \cup \{v\}$
    else
      $wp(t) := true$
      if $serp(u)$ **then**
        $Status(t) := \mathsf{abortsure}$; $urs(t) := \bot$

if $op = (\mathsf{rollback}, v), t)$ **then**
  if $Status(t) = \mathsf{commitsure}$ **then return** $\bot$
  if $v \notin ws(t)$ **then return** $\bot$
  if $wp(u)$ **then**
    $Status(u) := \mathsf{abortsure}$; $urs(u) := \bot$
  $ws(t) := ws(t) \setminus \{v\}$
  $Status(t) := \mathsf{abortsure}$; $urs(t) := \bot$

Algorithm 1 Continued.

nondetTMSpec($\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle, op$)

if $op = (\mathsf{rfin}, t)$ **then**
  if $urs(t) = \bot$ **then** return $\bot$ **else** $v := urs(t)$
  if $Status(t) = \mathsf{abortsure}$ **then** return $\bot$
  $rs(t) := rs(t) \cup \{v\}$; $urs(t) := \bot$
  if $wp(t)$ **then**
    if $serp(u)$ **then** return $\bot$ **else** $serp(t) := true$
    if $Status(u) = \mathsf{abortsure}$ **then** return $\bot$
    if $wp(u)$ **then** return $\bot$
    if $Status(u) = \mathsf{finished}$ **then**
      $Status(u) := \mathsf{commitsure}$
      if $urs(u) = v$ for some variable $v \in V$ **then**
        $rs(u) := rs(u) \cup \{v\}$
  if $rp(u)$ **then**
    if $serp(t)$ **then** return $\bot$ **else** $serp(u) := true$

if $op = (\varepsilon, t)$ **then**
  if $SerStatus(t) = true$ **then** return $\bot$
  $SerStatus(t) := true$
  if $SerStatus(u) = false$ **then**
    if $serp(t)$ **then** return $\bot$
    $serp(u) := true$
    if $wp(t) = true$ **then**
      $Status(t) := \mathsf{abortsure}$; $urs(t) := \bot$

if $op = (\mathsf{commit}, t)$ **then**
  if $SerStatus(t) \neq true$ **then** return $\bot$
  if $Status(t) = \mathsf{abortsure}$ **then** return $\bot$
  if $urs(t) \neq \bot$ **then** return $\bot$
  if $rp(t)$ **then**
    if $serp(u) = true$ **then**
      $Status(u) := \mathsf{abortsure}$; $urs(u) := \bot$
  if $serp(t)$ **then**
    $prs(u) := prs(u) \cup ws(t) \cup prs(t)$
    $pws(u) := pws(u) \cup ws(t) \cup rs(t) \cup pws(t)$
  $ResetState(t)$
  if $SerStatus(u)$ **then** $serp(t) := true$

if $op = (\mathsf{abort}, t)$ **then**
  if $SerStatus(t) = false$ **then** return $\bot$
  if $ws(t) \neq \emptyset$ **then** return $\bot$
  $ResetState(t)$
  if $SerStatus(u) = true$ **then** $serp(t) := true$
return $\langle Status, SerStatus, rs, ws, urs, prs, pws, wp, rp, serp\rangle$

3. The serialization status of x is *true* in a run $r' = r \cdot op$ if
   a. the serialization status of x in r is *true*, and op is not a commit or an abort of x, or
   b. op is a serialize of transaction x

4. The status of x is commitsure in a run $r' = r \cdot op$ if
   a. the status of x is commitsure in r and op is not a commit
   b. the status of x is finished in r and op is a store to v by y and x stores to v in r
   c. the status of x is finished in r and the status of y is commitsure and x stores to v in r and op is a load of v by y
   d. the status of x is finished in r and x stores to v and later y loads v in r and op is a finish of the load of v by y
5. The status of x is abortsure in a run $r' = r \cdot op$ if the status of x is not commitsure in r and one of the following holds:
   a. op is a store of a variable v by y and y serializes before x and x has an unused load of v in r
   b. op is a load of a variable v by x such that y stores to v in the run r and y serializes after x
   c. op is a rollback of v by y and x loads v after y stores to v in r
   d. op is a rollback of v by x and x stores to v in r
   e. op is a serialize of x and y is unserialized and there exists a variable v such that y stores to v and x loads v after y stores to v
   f. op is a load of a variable v by x and v is in the prohibited read set of x
6. The serialization predecessor *serp* of x is *true* in run $r' = r \cdot op$ if:
   a. the serialization predecessor of x is true in r and op is not a commit or an abort of transaction y
   b. op is a store of v by transaction x and y stores to v in r
   c. op is a store of v by x and y has a used load of v in r
   d. op is a store of v by x and the status of y is commitsure and y loads v
   e. op is a load of v by x and the status of x is commitsure in r and y stores to v in r
   f. op is a serialize of transaction y and the serialization status of x is false
   g. op is a finish of a read by transaction x and y stores to v in r, and later x loads v in r
   h. op is a finish of a read by transaction y and y loads v in r, and later x stores to v in r
7. The serialization predecessor of the transaction following x in the thread of x is *true* in a run $r' = r \cdot op$ if op is a commit or abort of x and the serialization status of y is *true*
8. Given a run r produced by *Spec* and an operation op of transaction x, the run $r' = r \cdot op$ is produced by *Spec* if the following hold:
   a. if the status of x is abortsure, then op is an abort, a rollback, or a serialize
   b. if op is a store of v, then x has no unused load in r and v is not in the prohibited write set of x
   c. if op is a rollback of v, then the status of x is not commitsure and x stores to v in r
   d. if op is a load of v, then x has no unused load in r
   e. if op is a load of v and v is in the prohibited read set of x, then status of x is not commitsure
   f. if op is a finish of a read, then there is an unused load by x and the status of x is not abortsure in r
   g. if op is a commit, then the serialization status of x is true and all loads by x in r are used
   h. if op is an abort, then there does not exist a variable v such that x stores to v and x does not rollback v in r
   i. if op is an abort, then the serialization status of x is true
   j. if op is a serialize, then the serialization status of x is false
   k. if the serialization predecessor of x is true, then the serialization predecessor of y is false in r'

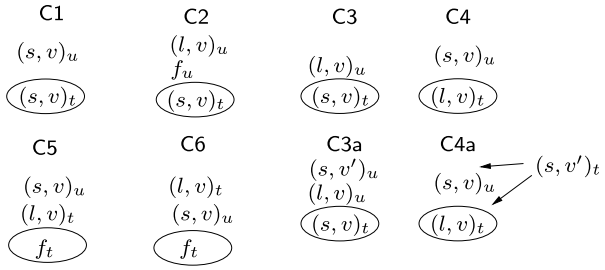

Fig. 3 The operations inside ovals are disallowed by the TM specification for opacity. An *arrow* represents different possible positions for a command to occur in a given condition. We write s for store, l for load, c for commit, b for rollback, and f for rfin. We write the operation $((s,v),t)$ as $(s,v)_t$. Thread t executes transaction x and thread u executes transaction y. For conditions C1–C5, the transaction y must serialize before x. For condition C6, the transaction x must serialize before y. For condition C3, y must commit in every extension of the run. For condition C4, x must commit in every extension of the run. Conditions C3a and C4a pertain to the nondeterministic TM specification for opacity without rollbacks, as described in the appendix

Using the above rules of construction, we now prove the correctness of the nondeterministic TM specification for opacity.

3.1.2 Correctness

Theorem 1 *Given a history h on 2 threads and k variables, h is opaque if and only if $h \in L(Spec)$.*

Proof We say that a transaction x must serialize before a transaction y in a run r if one of the following holds:

- x and y have final stores to a variable v and y stores to v after x stores to v
- the serialize of x occurs before the serialize of y in r
- x stores to v and y has a used load of v, where y loads v after x stores to v
- x has a used load of v and y stores v, where x loads v before y stores to v

Note that from rule 4, 8.c, and 8.h, a transaction x that stores to a variable v must commit in every extension of r if one of the following holds:

- there exists a transaction y such that y stores to v after x stores to v and before x rolls back
- there exists a transaction y such that y loads v after x stores to v, and the read of v is finished by y
- there exists a transaction y such that y must commit, and y loads v after x stores to v

Note that for two unfinished transactions x, y in a run r, if y must serialize before x, then the serialization predecessor of x is true in r.

Now, we note that the TM specification *Spec* for opacity gives the largest set R of runs such that for every run r produced by the TM specification, for every transaction x in r, the following conditions hold (conditions C1–C6 are graphically shown in Fig. 3):

C1. x does not store to a variable v if there exists a transaction y such that y must serialize after x and y stores to v and y does not rollback its store to v (from rules 1, 6.b, and 8.k)

C2. x does not store to a variable v if there exists a transaction y such that y must serialize after x and y has a used load of v (from rules 1, 6.c, and 8.k)

C3. x does not store to a variable v if there exists a transaction y such that y must serialize after x and y has a load of v and y must commit in every extension of r (from rules 1, 6.d, and 8.k)
C4. x does not load a variable v if there exists a transaction y such that y must serialize after x and y stores to v and x must serialize in every extension of r (from rules 2, 6.e, 8.e, and 8.k)
C5. x does not finish the read of a variable v if there exists a transaction y such that y must serialize after x and y stores to v before x loads v (from rules 2, 5.b, 5.f, 6.g, 8.f, and 8.k)
C6. x does not finish the read of a variable v if there exists a transaction y such that y must serialize before x and y stores to v after x loads v (from rules 5.a, 6.h, and 8.f)
C7. x serializes at most once (from rules 3 and 8.j)
C8. if x is a finished transaction, then x serializes exactly once (from rules 3, 8.g, 8.i, and 8.j)
C9. x contains a rollback of v only if the transaction consists of a store to v before the rollback (from rule 8.c)
C10. after a rollback of a variable v in x, the only possible instruction in x is a rollback of another variable or a serialize or an abort (from rules 5.d and 8.a)
C11. if x is an aborting transaction, then every aborted transaction rollbacks all the stores before aborting (from rule 8.h)
C12. x does not serialize if there exists a transaction y such that y is unserialized and y must serialize before x (from rules 6.f and 8.k)

Let h be an opaque history on 2 threads and k variables. As h is opaque, there is a sequential history h_s such that h_s is strictly equivalent to h. Let the transactions in the sequential history h_s be given by the sequence $x_1 \ldots x_n$ of transactions. We claim that there exists a run r of the TM specification *Spec* such that h is the corresponding history of r. As h_s is strictly equivalent to h, we know that for every pair x_i, x_j of transactions in h such that $i < j$, the following are not true:

– x_i loads v after a final store to v by x_j, and x_i finishes the read
– x_i is a committing transaction and x_i loads v after a final store by x_j to v
– x_i and x_j have a final store to v and x_i stores to v after x_j stores to v
– x_i stores to v and x_j loads v before the store to v, and the read of v is finished by x_j

These conditions are equivalent to the conditions C1–C6. Moreover, h is well-formed. This is equivalent to the conditions C9–C11. Thus, we know that there exists a run r of the TM specification *Spec*, where the order of serialization of transactions is the same as $x_1 \ldots x_n$.

Conversely, let r be a run produced by the nondeterministic TM specification *Spec*. Let h be the corresponding history to the run r. We know from conditions C7 and C8 that every transaction serializes at most once in the run, and every finished transaction serializes exactly once in the run. Let h_s be a sequential history such that (i) a transaction x appears before a transaction y in h_s if x must serialize before y in r, (ii) all other transactions appear in an arbitrary order later in r, and (iii) for all threads, the thread projection of h_s is equivalent to the thread projection of h. The conditions C1–C6 guarantee that for every pair op_i, op_j of operations in h if op_i and op_j conflict and $i < j$, then op_i occurs before op_j in h_s. Note that the order of serialization in r respects the real time order of the transactions in h, that is, if a transaction x finishes before a transaction y starts, then x serializes before y in r. Thus, h_s is strictly equivalent to h. Hence, every history in $L(Spec)$ is opaque. □

Algorithm 2 The deterministic STM specification for opacity

detTMSpec($\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle, op$)

if $op = ((\mathsf{store}, v), t)$ **then**
  if $Status(t) = \mathsf{abortsure}$ **then** return $\bot$
  if $urs(t) \neq \bot$ **then** return $\bot$
  if $v \in pws(t)$ **then return** $\bot$
  $ws(t) := ws(t) \cup \{v\}$
  if $Status(t) = \mathsf{finished}$ **then**
    if $Status(u) = \mathsf{pending}$ **then** $hp(t) := true$
    if $Status(u) = \mathsf{commitsurepending}$ **then** $sp(t) := true$
    $Status(t) := \mathsf{started}$
  if $v \in ws(u)$ **then**
    if $wp(u)$ **or** $hp(u)$ **then** return $\bot$
    if $Status(u) = \mathsf{abortsure}$ **then** return $\bot$
    if $Status(u) = \mathsf{commitimpossible}$ **then** return $\bot$
    if $Status(u) = \mathsf{pending}$ **then**
      $Status(u) := \mathsf{commitsurepending}$
      if $urs(u) = v$ for some variable $v \in V$ **then** $rs(u) := rs(u) \cup \{v\}$
      if $wp(u)$ **or** $rp(u)$ **then** $sp(u) := true$
    if $Status(u) = \mathsf{started}$ **then**
      $Status(u) := \mathsf{commitsure}$
      if $urs(u) = v$ for some variable $v \in V$ **then** $rs(u) := rs(u) \cup \{v\}$
      if $wp(u)$ **or** $rp(u)$ **then** $sp(u) := true$
    $sp(t) := true$
  if $v \in rs(u)$ **then**
    $sp(t) := true$
    if $wp(u)$ **then** $Status(u) := \mathsf{abortsure}$; $urs(u) := \emptyset$
  if $v \in urs(u)$ **then**
    $rp(t) := true$
    if $sp(u)$ **or** $hp(u)$ **or** $wp(u)$ **then**
      $Status(u) := \mathsf{abortsure}$; $urs(u) := \bot$
  if $sp(u)$ **and** $sp(t)$ **then** return $\bot$

3.2 A deterministic TM specification

We define the *deterministic TM specification for opacity* $Spec^d$ for two threads as the tuple $\langle Q, q_{init}, \delta\rangle$. A state $q \in Q$ is a 10-tuple $\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle$, where $Status : T \to \{\mathsf{finished}, \mathsf{pending}, \mathsf{commitsurepending}, \mathsf{abortsure}, \mathsf{commitsure}\}$ is the status, $rs : T \to 2^V$ is the read set, $ws : T \to 2^V$ is the write set, $urs : T \to \{\bot\} \cup V$ is the unfinished read set, $prs : T \to 2^V$ is the prohibited read set, $pws : T \to 2^V$ is the prohibited write set, $hp : T \to \{true, false\}$ is the hidden predecessor (to due a transaction already committed) flag, $wp : T \to \{true, false\}$ is the write predecessor flag, $rp : T \to \{true, false\}$ is the read predecessor flag, and $sp : T \to \{true, false\}$ is the strong predecessor flag for the threads. The initial state $q_{init} = \langle Status_0, rs_0, ws_0, urs_0, prs_0, pws_0, hp_0, wp_0, rp_0, sp_0\rangle$, where $Status_0(t) = \mathsf{finished}$, $urs_0(t) = \bot$, $hp_0(t) = wp_0(t) = rp_0(t) = sp_0(t) = false$, and $rs_0(t) = ws_0(t) = prs_0(t) = pws_0(t) = \emptyset$ for both threads. The transition relation of the deterministic TM specification is obtained using Algorithm 2. Given a state q, the procedure $ResetState(q, t)$ makes the following updates: (i) sets $Status(t)$ to $\mathsf{finished}$, (ii) sets $urs(t)$ to $\bot$, (iii) sets $rs(t)$, $ws(t)$, $prs(t)$, and $pws(t)$ to $\emptyset$, (iv) sets

Algorithm 2 Continued.

detTMSpec($\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle, op$)

if $op = (\text{rollback}, v), t)$ **then**
  if $Status(t) \in \{\text{commitsure}, \text{commitsurepending}\}$ **then** return $\bot$
  if $v \notin ws(t)$ **then** return $\bot$
  if $wp(u)$ **then**
    $Status(u) := \text{abortsure}$; $urs(u) := \bot$
  $ws(t) := ws(t) \setminus \{v\}$
  $Status(t) := \text{abortsure}$; $urs(t) := \bot$

if $op = (\text{rfin}, t)$ **then**
  if $urs(t) = \bot$ **then** return $\bot$
  $v := urs(t)$
  if $v \in prs(t)$ **and** $Status(t) = \text{commitimpossible}$ **then** return $\bot$
  $rs(t) := rs(t) \cup \{v\}$; $urs(t) = \bot$
  if $rp(u)$ **then** $sp(u) := true$
  if $Status(t) = \text{pending}$ **or** $Status(t) = \text{commitsurepending}$ **then**
    $sp(u) := true$
  if $wp(t)$ **then**
    if $Status(u) \in \{\text{abortsure}, \text{commitimpossible}\}$ **then** return $\bot$
    if $Status(u) = \text{pending}$ **then**
      $Status(u) := \text{commitsurepending}$
      if $urs(u) = v$ for some variable $v \in V$ **then** $rs(u) := rs(u) \cup \{v\}$
      if $wp(u)$ **or** $rp(u)$ **then** $sp(u) := true$
    if $Status(u) = \text{started}$ **then**
      $Status(u) := \text{commitsure}$
      if $urs(u) = v$ for some variable $v \in V$ **then** $rs(u) := rs(u) \cup \{v\}$
      if $wp(u)$ **or** $rp(u)$ **then** $sp(u) := true$
    $sp(t) := true$
  if $sp(u)$ **and** $sp(t)$ **then** return $\bot$

if $op = ((\text{load}, v), t)$ **then**
  if $Status(t) = \text{abortsure}$ **then** return $\bot$
  if $urs(t) \neq \bot$ **then** return $\bot$
  if $Status(t) = \text{finished}$ **then**
    if $Status(u) = \text{pending}$ **then** $hp(t) := true$
    if $Status(u) = \text{commitsurepending}$ **then** $sp(t) := true$
    $Status(t) := \text{started}$
  if $v \in prs(t)$ **then**
    if $Status(t) \in \{\text{commitsure}, \text{commitsurepending}\}$ **then** return $\bot$
    $Status(t) := \text{commitimpossible}$
  $urs(t) := v$
  if $Status(t) \in \{\text{commitsure}, \text{commitsurepending}\}$ **then** $rs(t) := rs(t) \cup \{v\}$
  if $v \in ws(u)$ **then**
    if $Status(t) \in \{\text{commitsure}, \text{commitsurepending}\}$ **then**
      $rs(t) := rs(t) \cup \{v\}$
      if $Status(u) \in \{\text{abortsure}, \text{commitimpossible}\}$ **then** return $\bot$
    if $Status(u) = \text{pending}$ **then**
      $Status(u) := \text{commitsurepending}$
      if $urs(u) = v$ for some variable $v \in V$ **then** $rs(u) := rs(u) \cup \{v\}$
      if $wp(u)$ **or** $rp(u)$ **then** $sp(u) := true$

Algorithm 2 Continued.

detTMSpec($\langle Status, rs, ws, urs, prs, pws, hp, wp, rp, sp\rangle, op$)

```
        if Status(u) = started then
          Status(u) := commitsure
          if urs(u) = v for some variable v ∈ V then rs(u) := rs(u) ∪ {v}
          if wp(u) or rp(u) then sp(u) := true
        sp(t) := true
        else
          wp(t) := true
          if sp(u) then
            Status(t) := abortsure;    urs(t) :=⊥
    if sp(u) and sp(t) then return ⊥

  if op = (commit, t) then
    if Status(t) ∈ {abortsure, commitimpossible} then return ⊥
    if urs(t) ≠⊥ then return ⊥
    if hp(t) then
      if sp(u) then
        if urs(u) ≠⊥ then
          Status(u) := abortsure;    urs(u) := ∅
        else Status(u) = commitimpossible
    if hp(t) or rp(t) or sp(t) then
      if Status(u) = started then Status(u) = pending
      if Status(u) = commitsure then Status(u) = commitsurepending
      prs(u) := prs(u) ∪ ws(t) ∪ prs(t)
      pws(u) := pws(u) ∪ ws(t) ∪ rs(t) ∪ pws(t)
    if sp(u) and sp(t) then return ⊥
    ResetState(t)

  if op = (abort, t) then
    if ws(t) ≠ ∅ then return ⊥
    ResetState(t)
  return ⟨Status, rs, ws, urs, prs, pws, hp, wp, rp, sp⟩
```

$hp(t)$, $rp(t)$, $wp(t)$, $hp(u)$, $rp(u)$, $wp(u)$, and $sp(u)$ to *false*. As in the nondeterministic TM specification, t refers to the thread taking the step, and u refers to the other thread.

Correctness We use an antichain based tool [10] to prove that for two threads and two variables, the language of the deterministic TM specification is equivalent to the language of the nondeterministic TM specification. The nondeterministic TM specification has 155,000 states, while the deterministic TM specification has 46,000 states. The execution time for checking equivalence of the two specifications using the antichain based tool [10] on an Opteron machine with 2.66 GHz processors and 16 GB RAM is around two hours. This high execution time is mostly due to the high memory consumption. The check consumes around 15 GB RAM, and thus most of the time is spent in swapping memory. The process execution time is around ten minutes.

The manual proof for the intuitive nondeterministic TM specification and the automated language equivalence check with the deterministic TM specification allow us to claim that the deterministic TM specification accepts exactly the set of opaque histories.

4 Memory models

A *memory model* is a function $M : Inst \times Inst \to \{N, E, Y\}$. For all instructions $inst_1$, $inst_2 \in Inst$, when $inst_1$ is immediately followed by $inst_2$, we have: (i) if $M(inst_1, inst_2) = N$, then M imposes a strict order between $inst_1$ and $inst_2$, (ii) if $M(inst_1, inst_2) = E$, then M allows to eliminate the instruction $inst_2$, and (iii) if $M(inst_1, inst_2) = Y$, then M allows to reorder $inst_1$ and $inst_2$. The case (ii) allows us to model store load forwarding using store buffers, and the case (iii) allows us to model reordering of instructions. Our formalism can capture many hardware memory models. But, our formalism cannot capture some common compiler optimizations like irrelevant read elimination, and thus disallows many software memory models (like the Java memory model [29]). We specify different memory models in our framework. These memory models are chosen to illustrate different levels of relaxations generally provided by the hardware. Let $\mathbf{M}$ be the set of all memory models.

Sequential consistency Sequential consistency does not allow any pair of instructions to be reordered. *Sequential consistency* [26] is specified by the memory model M_{sc}. We have $M_{sc}(inst_1, inst_2) = N$ for all instructions $inst_1, inst_2 \in Inst$.

Total store order Total store order (TSO) relaxes the order of a store followed by a load to a different address. But, TSO enforces a strict order on the stores (and hence the name). TSO allows a load which follows a store to the same address to be eliminated. TSO [42] is given by the memory model M_{tso} such that for all memory instructions $inst_1, inst_2 \in Inst$, (i) if $inst_1 = \langle \mathsf{store}\ a \rangle$ and $inst_2 = \langle \mathsf{load}\ a' \rangle$ such that $a \neq a'$, then $M_{tso}(inst_1, inst_2) = Y$, (ii) if $inst_1 \in \{\langle \mathsf{store}\ a \rangle, \langle \mathsf{cas}\ a \rangle\}$ and $inst_2 = \langle \mathsf{load}\ a \rangle$, then $M_{tso}(inst_1, inst_2) = E$, (iii) else $M_{tso}(inst_1, inst_2) = N$.

Partial store order Partial store order (PSO) is similar to TSO, but further relaxes the order of stores. PSO [42] is specified by M_{pso}, such that for all memory instructions $inst_1, inst_2 \in Inst$, (i) if $inst_1 = \langle \mathsf{store}\ a \rangle$ and $inst_2 \in \{\langle \mathsf{load}\ a' \rangle, \langle \mathsf{store}\ a' \rangle, \langle \mathsf{cas}\ a' \rangle\}$ such that $a \neq a'$, then $M_{pso}(inst_1, inst_2) = Y$, (ii) if $inst_1 \in \{\langle \mathsf{store}\ a \rangle, \langle \mathsf{cas}\ a \rangle\}$ and $inst_2 = \langle \mathsf{load}\ a \rangle$, then $M_{pso}(inst_1, inst_2) = E$, (iii) else $M_{pso}(inst_1, inst_2) = N$.

Relaxed memory order Relaxed memory order (RMO) relaxes the order of instructions even more than PSO. RMO allows to reorder a load with a following load or a following store to a different address. RMO [42] is specified by M_{rmo}, such that for all memory instructions $inst_1, inst_2 \in Inst$, (i) if $inst_1 \in \{\langle \mathsf{load}\ a \rangle, \langle \mathsf{store}\ a \rangle, \langle \mathsf{cas}\ a \rangle\}$ and $inst_2 \in \{\langle \mathsf{load}\ a'' \rangle, \langle \mathsf{store}\ a' \rangle, \langle \mathsf{cas}\ a' \rangle\}$ such that $a \neq a'$ (note that a'' can be same as a), then $M_{rmo}(inst_1, inst_2) = Y$, (ii) if $inst_1 \in \{\langle \mathsf{store}\ a \rangle, \langle \mathsf{cas}\ a \rangle\}$ and $inst_2 = \langle \mathsf{load}\ a \rangle$, then $M_{rmo}(inst_1, inst_2) = E$, (iii) else $M_{rmo}(inst_1, inst_2) = N$. Note that at the level of instruction streams, we do not capture control/data dependence. Rather, we allow RMO to reorder any pair of instructions.

4.1 Memory-model sensitive semantics

Intuitively, capturing a relaxed memory model requires us to defer statements across following statements, unless the memory model guarantees an ordering. So, RML maintains

as part of the state, a queue of statements whose execution has been deferred. When a statement with a memory instruction is encountered, RML inserts the statement in the queue of deferred statements. However, the relaxations allowed by the memory model allow to insert the statement at multiple places in the queue. Thus, we obtain multiple transitions from the original state on a statement with a memory instruction, where each destination state differs only in the queue of deferred statements. When RML encounters a store (resp. load) fence, RML dequeues statements for execution, until the queue has no store (resp. load) instructions. We now formalize the semantics of RML.

Let G and L be the set of global and local addresses respectively. Let $Idx \subseteq L$ be the set of local index addresses. Consider a particular thread t. Let $\sigma : G \cup L \to \mathbb{N}$ be a *valuation* of the global addresses, and the local addresses of thread t. Let Σ be the set of all valuations. Note that the syntax of RML is defined in a way that the value of an index variable *idx* may not depend on the queue of deferred statements. Given a global location g and a valuation σ, we write $[\![g]\!]_\sigma \in G$ to denote the global address represented by g in valuation σ. Similarly, we write $[\![l]\!]_\sigma \in L \setminus Idx$ (resp. $[\![idx]\!]_\sigma \in Idx$) to denote the local address (resp. local index address) represented by a local location l (resp. index variable *idx*) in valuation σ.

Let $\gamma : S_M \times \Sigma \to Inst \cup \{\mathsf{skip}\}$ be a mapping function for memory statements, that for a given memory statement and a valuation, gives the generated hardware instruction or the skip instruction if no hardware instruction is generated. For example, we have $\gamma(g := e, \sigma) = \langle \mathsf{store}\ [\![g]\!]_\sigma \rangle$ in valuation σ, as the statement $g := e$ causes a store to the global address represented by g in valuation σ. The statement rollback $g := e$ is physically a store instruction, as a rollback undoes the effect of a previous store instruction. We define a *local-variables* function *lvars* such that given an expression e and a valuation σ, we have $lvars(e, \sigma)$ as the smallest set of local addresses in L such that if the location l (resp. index variable *idx*) appears in e, then the address $[\![l]\!]_\sigma$ is in $lvars(e, \sigma)$ (resp. $[\![idx]\!]_\sigma$ is in $lvars(e, \sigma)$). We define a *write-locals* function $lw : S_M \times \Sigma \to 2^L$ and a *read-locals* function $lr : S_M \times \Sigma \to 2^L$ to obtain the written and read local addresses in a statement respectively. Table 1 gives the formal definitions of the functions γ, lw, and lr.

We now describe when two memory statements can be reordered in a given valuation under a given memory model. Let $\mathbf{M}$ be the set of all memory models. Let $R : S_M \times S_M \times \Sigma \times \mathbf{M} \to \{true, false\}$ be a *reordering* function such that $R(s_1, s_2, \sigma, M) = true$ if the following conditions hold: (i) either $\gamma(s_1, \sigma) = \mathsf{skip}$ or $\gamma(s_2, \sigma) = \mathsf{skip}$, or $M(\gamma(s_1, \sigma), \gamma(s_2, \sigma)) = Y$, (ii) $lw(s_1, \sigma) \cap lr(s_2, \sigma) = \emptyset$, (iii) $lw(s_1, \sigma) \cap lw(s_2, \sigma) = \emptyset$, and (iv) $lr(s_1, \sigma) \cap lw(s_2, \sigma) = \emptyset$. Here, the first condition restricts reorderings to those allowed by the memory model, and the remaining conditions check for data dependence between the statements. To defer memory statements and execute them in all different ways as allowed by the relaxed memory model, we define a model-dependent enqueue function. This function takes as input the current valuation, the current sequence of deferred statements, a statement to defer, and a memory model, and produces the set of new possible sequences of deferred statements. We define the *enqueue* function $Enq : S_M^* \times S_M \times \Sigma \times \mathbf{M} \to 2^{S_M^*}$ such that given a sequence $d = s_1 \ldots s_n$ of memory statements, a statement s, a valuation σ, and a memory model M, the function $Enq(d, s, \sigma, M)$ is the largest set such that (i) $s_1 \ldots s_k \cdot s \cdot s_{k+1} \ldots s_n \in Enq(d, s, \sigma, M)$ if for all i such that $k < i \leq n$, we have $R(s_i, s, \sigma, M) = true$, and (ii) if s is of the form $l := g$, then $s_1 \ldots s_k \cdot (l := e) \cdot s_{k+1} \ldots s_n \in Enq(d, s, \sigma, M)$ if for all i with $k < i \leq n$, we have $R(s_i, s, \sigma, M) = true$, and $M(\gamma(s_k, \sigma), \gamma(s, \sigma)) = E$ where (a) if s_k is $g := f$, then $e = f$, (b) if s_k is $m := g$ or $m := \mathsf{cas}(g, e_1, e_2)$, then $e = m$. Note that the definition of the reordering function restricts the reordering of control and data-dependent statements. Thus, our model of RMO slightly differs from the definition of the RMO model in the sense that we impose an order on control dependent load instructions. Similarly, the enqueue function restricts

Table 1 The formal definitions of the functions γ, lw, and lr for a statement s in a valuation σ

Statement s	$\gamma(s,\sigma)$	$lw(s,\sigma)$	$lr(s,\sigma)$
$g := e$	$\langle \mathsf{store}\ [\![g]\!]_\sigma \rangle$	$\emptyset$	$lvars(e,\sigma)$
$l := g$	$\langle \mathsf{load}\ [\![g]\!]_\sigma \rangle$	$\{[\![l]\!]_\sigma\}$	$\emptyset$
$l := e$	skip	$\{[\![l]\!]_\sigma\}$	$lvars(e,\sigma)$
$l := \mathsf{cas}(g, e_1, e_2)$	$\langle \mathsf{cas}\ [\![g]\!]_\sigma \rangle$	$\{[\![l]\!]_\sigma\}$	$lvars(e_1,\sigma) \cup lvars(e_2,\sigma)$
rollback $g := e$	$\langle \mathsf{store}\ [\![g]\!]_\sigma \rangle$	$\emptyset$	$lvars(e,\sigma)$
$idx := c$	skip	$\{[\![idx]\!]_\sigma\}$	$lvars(c,\sigma)$

the elimination of only load instructions. While this is sufficient to model many hardware memory models, we cannot capture coalesced stores or redundant store elimination.

Given a valuation σ, a program p, and a sequence d of deferred statements, we define a predicate $allowDequeue(\sigma, d, p)$ to be *true* if (i) p is of the form (**while** e **do** $p_1; p'$) or (**if** e **then** p_1 **else** $p_2; p'$) for some programs $p_1, p_2, p' \in P$, and there exists a memory statement s in d such that $lw(s,\sigma) \cap lvars(e,\sigma) \neq \emptyset$, or (ii) p is a store fence and there exists a statement s of the form $g := l$ or $l := \mathsf{cas}(g, e, e)$ in d, or (iii) p is a load fence and there exists a statement s of the form $l := g$ or $l := \mathsf{cas}(g, e, e)$ in d.

Conditionals and loops When an RML program reaches a condition or a loop, it requires the condition to be evaluated. RML first checks whether the local variables appearing in the condition are modified in any deferred statement in the queue. If this is not the case, the execution is governed by the following rules.

$$\frac{\sigma[e] \neq 0 \quad allowDequeue(\sigma, d, p) = false \quad p = \textbf{if}\ e\ \textbf{then}\ p_1\ \textbf{else}\ p_2;\ p'}{\langle p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p_1; p',\ \sigma,\ d\rangle} \quad \text{(IF TRUE)}$$

$$\frac{\sigma[e] = 0 \quad allowDequeue(\sigma, d, p) = false \quad p = \textbf{if}\ e\ \textbf{then}\ p_1\ \textbf{else}\ p_2;\ p'}{\langle p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p_2; p,\ \sigma,\ d\rangle} \quad \text{(IF FALSE)}$$

$$\frac{\sigma[e] \neq 0 \quad allowDequeue(\sigma, d, p) = false \quad p = \textbf{while}\ e\ \textbf{do}\ p_1;\ p'}{\langle p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p_1; p,\ \sigma,\ d\rangle} \quad \text{(WHILE TRUE)}$$

$$\frac{\sigma[e] = 0 \quad allowDequeue(\sigma, d, p) = false \quad p = \textbf{while}\ e\ \textbf{do}\ p_1;\ p'}{\langle p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p',\ \sigma,\ d\rangle} \quad \text{(WHILE FALSE)}$$

Index variable update As the value of an index variable does not depend on the deferred statements, it is straightforward to modify the valuation according to the variable update.

$$\frac{}{\langle idx := c; p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p,\ \sigma[idx/c],\ d\rangle} \quad \text{(INDEX)}$$

Fences When an RML program encounters a store (resp. load) fence, we ensure that there is no store or cas (resp. load or cas) instruction in the queue of deferred statements.

$$\frac{allowDequeue(\sigma, d, \mathsf{stfence}) = false}{\langle \mathsf{stfence};\ p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p,\ \sigma,\ d\rangle} \quad \text{(STORE FENCE)}$$

$$\frac{allowDequeue(\sigma, d, \mathsf{ldfence}) = false}{\langle \mathsf{ldfence};\ p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p,\ \sigma,\ d\rangle} \quad \text{(LOAD FENCE)}$$

TM specific A commit and an abort instruction behave like store fences. This is to avoid instructions of two transactions from the same thread to interleave with each other. Moreover, an rfin instruction behaves like a load fence. This ensures that during the transactional read of a variable, the variable is loaded before the read is declared as finished.

$$\frac{\begin{array}{c} s \in \{\mathsf{commit}, \mathsf{abort}\} \\ allowDequeue(\sigma, d, \mathsf{stfence}) = false \end{array}}{\langle s;\ p,\ \sigma,\ d\rangle \xrightarrow{s} \langle p,\ \sigma,\ d\rangle} \quad \text{(TRANSACTION END)}$$

$$\frac{allowDequeue(\sigma, d, \mathsf{ldfence}) = false}{\langle \mathsf{rfin};\ p,\ \sigma,\ d\rangle \xrightarrow{\mathsf{rfin}} \langle p,\ \sigma,\ d\rangle} \quad \text{(READ FINISH)}$$

Enqueue When RML encounters a memory instruction in the form of a load, store, compare-and-swap, or rollback instruction, RML enqueues the statement into the queue of deferred statements. Then, RML nondeterministically shuffles the statement in the queue according to the relaxations allowed by the underlying memory model M.

$$\frac{\begin{array}{c} d' \in Enq(d, s, \sigma, M) \\ s \in \{g := e, l := g, l := e, l := \mathsf{cas}(g, e_1, e_2), \mathsf{rollback}\ g := e\} \end{array}}{\langle s;\ p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p,\ \sigma,\ d'\rangle} \quad \text{(ENQUEUE)}$$

Dequeue A statement is executed by RML only under the circumstances that the value of the conditional variable depends on the contents on the queue, or RML reaches a store (resp. load) fence, and there is a store (resp. load) instruction in the queue. In this case, the first statement in the queue is dequeued, and the effect of the statement is made to the global and local variables as required.

$$\frac{\begin{array}{c} allowDequeue(\sigma, d, p_1) = true \\ \sigma[g] = c \qquad d = (l := g) \cdot d' \end{array}}{\langle p_1;\ p,\ \sigma,\ d\rangle \xrightarrow{\langle \mathsf{load}\ [\![g]\!]_\sigma \rangle} \langle p_1;\ p,\ \sigma[l/c],\ d'\rangle} \quad \text{(DEQUEUE LOAD)}$$

$$\frac{\begin{array}{c} allowDequeue(\sigma, d, p_1) = true \\ \sigma[e] = c \qquad d = (g := e) \cdot d' \end{array}}{\langle p_1;\ p,\ \sigma,\ d\rangle \xrightarrow{\langle \mathsf{store}\ [\![g]\!]_\sigma \rangle} \langle p_1;\ p,\ \sigma[g/c],\ d'\rangle} \quad \text{(DEQUEUE STORE)}$$

Fig. 4 An example of an RML program

```
initially, X1 := 0, X2 := 0, Y1 := 0, Y2 := 0
           r1 := 0, r2 := 0, r3 := 0, r4 := 0
if id = 0                          if id = 1
    X1 := 0                            X2 := 0
    Y1 := 0                            Y2 := 0
    r1 := Y2          ||               r2 := Y1
    r3 := X2                           r4 := X1
    X1 := 2                            X2 := 2
```

$$\frac{allowDequeue(\sigma, d, p_1) = true \quad \sigma[g] = c \quad \sigma[e_1] \neq c \quad d = (l := \mathsf{cas}(g, e_1, e_2)) \cdot d'}{\langle p_1; p,\ \sigma,\ d\rangle \xrightarrow{\langle \mathsf{load}\ [\![g]\!]_\sigma \rangle} \langle p_1; p,\ \sigma[l/c],\ d'\rangle} \quad \text{(DEQUEUE CAS FAILURE)}$$

$$\frac{allowDequeue(\sigma, d, p_1) = true \quad \sigma[g] = \sigma[e_1] = c \quad \sigma[e_2] = c' \quad d = (l := \mathsf{cas}(g, e_1, e_2)) \cdot d'}{\langle p_1; p,\ \sigma,\ d\rangle \xrightarrow{\langle \mathsf{cas}\ [\![g]\!]_\sigma \rangle} \langle p_1; p,\ \sigma[g/c'][l/c],\ d'\rangle}$$

(DEQUEUE CAS SUCCESS)

$$\frac{allowDequeue(\sigma, d, p_1) = true \quad \sigma[e] = c \quad d = (l := e) \cdot d'}{\langle p_1; p,\ \sigma,\ d\rangle \xrightarrow{\varepsilon} \langle p_1; p,\ \sigma[l/c],\ d'\rangle} \quad \text{(DEQUEUE LOCAL)}$$

$$\frac{allowDequeue(\sigma, d, p_1) = true \quad \sigma[e] = c \quad d = (\mathsf{rollback}\ g := e) \cdot d'}{\langle p_1; p,\ \sigma,\ d\rangle \xrightarrow{\mathsf{rollback}[\![g]\!]_\sigma} \langle p_1; p,\ \sigma[g/c],\ d'\rangle} \quad \text{(DEQUEUE ROLLBACK)}$$

4.2 Example execution in RML

Consider the RML program shown in Fig. 4. We discuss the operation of RML on this program under different memory models.

- *Sequential consistency* does not allow any reorderings. The example generates 247 states.[1]

  There are seven possible valuations for the tuple ⟨r1, r2, r3, r4⟩ of variables: ⟨1, 0, 2, 0⟩, ⟨1, 0, 1, 0⟩, ⟨1, 1, 2, 1⟩, ⟨1, 1, 1, 1⟩, ⟨1, 1, 1, 2⟩, ⟨0, 1, 0, 1⟩, ⟨0, 1, 0, 2⟩.
- *Total store order* allows to reorder a store followed by a read of a different variable, and also allows to eliminate a read of a variable following a store to the same variable. The example generates 943 states. Apart from the valuations allowed in sequential consistency, TSO allows one more valuation for ⟨r1, r2, r3, r4⟩: ⟨0, 0, 0, 0⟩.

[1] In this example, we define a state as a valuation of the global and local variables, the program counters, and the deferred statements for each thread.

– *Partial store order* further allows to reorder stores of different variables. The example generates 3382 states. Apart from the valuations allowed in TSO, PSO allows seven more valuation for $\langle \mathsf{r1}, \mathsf{r2}, \mathsf{r3}, \mathsf{r4} \rangle$: $\langle 1,0,0,0 \rangle$, $\langle 1,1,0,1 \rangle$, $\langle 1,1,0,2 \rangle$, $\langle 1,1,2,0 \rangle$, $\langle 1,1,1,0 \rangle$, $\langle 1,1,0,0 \rangle$, $\langle 0,1,0,0 \rangle$.
– *Relaxed memory order* further allows to reorder loads and a load followed by a store to a different variable. The example generates 26596 states. Apart from the valuations allowed in PSO, RMO allows one more valuation: $\langle 1,1,2,2 \rangle$.

We indeed produce these outcomes using our automated tool FOIL which we present in the next chapter. Some of the outcomes in the above examples are hard to reason about manually. That shows the importance of automated tools to reason about outcomes under relaxed memory models.

5 STM algorithms in RML

A state of a thread carries the information of the program currently being executed, the valuation of the local variables, the deferred statements of the thread, and the location of the transactional program. A *thread-local state* z_l^t of thread t is the tuple $\langle p^t, \sigma_L^t, D^t, loc^t \rangle$, where p^t is the current RML program being executed by thread t, $\sigma_L^t : L \cup Idx \to \mathbb{N}$ is the valuation of the local and index variables of thread t, D^t is the deferred statements of thread t, and $loc^t \in \mathbb{B}^*$ is the location of the transactional program θ^t. A *state* z of an STM algorithm with T threads is given by $\langle \sigma_G, z_l^1 \ldots z_l^T \rangle$, where $\sigma_G : G \to \mathbb{N}$ is the valuation of the global variables of the STM algorithm, and z_l^t is the thread-local state of thread t for $1 \leq t \leq T$. An STM algorithm A is a 4-tuple $\langle p_r, p_w, p_e, z_{init} \rangle$, where p_r, p_w, and p_e are RML programs, and z_{init} is the initial state of the STM algorithm. Moreover, we define a function $\alpha : C \to P$ that maps a transactional command to an RML program, such that $\alpha((\mathsf{read}, k)) = (v := k; p_r)$, $\alpha((\mathsf{write}, k)) = (v := k; p_w)$, and $\alpha(\mathsf{xend}) = p_e$.

5.1 Language of an STM algorithm

Let a *scheduler* σ on T be a function $\sigma : \mathbb{N} \to T$. Given a scheduler σ, a transactional program *prog*, and a memory model M, a *run* of an STM algorithm A is a sequence $\langle z_0, i_0 \rangle, \ldots \langle z_n, i_n \rangle$ such that $z_0 = z_{init}$, and for all j such that $0 \leq j < n$, if $z_j = \langle \sigma_G, z_l^1 \ldots z_l^T \rangle$ and $z_{j+1} = \langle \sigma'_G, z_l'^1 \ldots z_l'^T \rangle$, then (i) $\langle p, \sigma_G \cup \sigma_L, D \rangle \xrightarrow{i_{j+1}} \langle p', \sigma'_G \cup \sigma'_L, D' \rangle$ is a step of RML with memory model M, and (ii) for all threads $t \neq \sigma(j)$, we have $z_l'' = z_l^t$, and (iii) for thread $t = \sigma(j)$, we have $z_l'' = \langle p'', \sigma'_L, D' \rangle$, where (a) if $i_{j+1} \in \{\mathsf{rfin}, \mathsf{commit}\}$, then $p'' = \alpha(\theta(loc \cdot 1))$, (b) else if $i_{j+1} = \mathsf{abort}$, then $p'' = \alpha(\theta(loc \cdot 0))$, (c) else $p'' = p'$. A run $\langle z_{init}, i_0 \rangle, \ldots, \langle z_n, i_n \rangle$ of an STM algorithm A produces a history h such that h is the longest subsequence of operations in $i_1 \ldots i_n$. The *language* $L(A, M)$ of an STM algorithm A under a memory model M is the set of all histories h where there exists a multi-threaded transactional program *prog* and a scheduler σ such that h can be produced by A on *prog* with σ under M. An STM algorithm A is safe for property π under a memory model M if every history in the language of A under M is included in π.

5.2 Examples

We now describe three STM algorithms, DSTM, TL2, and McRT-STM in RML. We present the RML programs p_r, p_w, and p_e for each STM algorithm. We also give the initial valuation

Algorithm 3 The TL2 algorithm in RML

```
01 program p_a :
02 u := 0
03 while u < V do
04   u := u + 1;
05   if owner[u] = self then
06     owner[u] := 0;
07   rs[u] := 0; ws[u] := 0
08 lclock := 0
09 abort
```

```
01 program p_w :
02 if lclock = 0 then
03   lclock := clk;
04 ws[v] := 1;
```

```
01 program p_r :
02 if lclock = 0 then
03   lclock := clk;
04 if ws[v] = 0 then
05   l := owner[v];
06   if l ≠ 0 then p_a
07   l := g[v];
08   lver[v] := version[v];
09   if lclock ≠ lver[v] then p_a
10   rs[v] := 1;
11   rfin
```

```
01 program p_e :
02 u := 0;
03 while u < V do
04   u := u + 1;
05   if (ws[u] = 1) then
06     l := cas(owner[u], 0, self);
07     if l ≠ self then p_a
08 l := 0;
09 while l ≠ lclock + 1 do
10   lclock := clk
11   l := cas(clk, lclock, lclock + 1);
12 u := 0;
13 while u < V do
14   u := u + 1;
15   if rs[u] = 1 then
16     rs[u] := 0;
17     l := owner[u];
18     c := version[u];
19     if c ≠ lver[u] then p_a
20     if l ≠ 0 then p_a
21 u := 0;
22 while u < V do
23   u := u + 1;
24   if ws[u] = 1 then
25     version[u] := lclock;
26     g[u] := l;
27 u := 0;
28 while u < V do
29   u := u + 1;
30   if ws[u] = 1 then
31     owner[u] := 0;
32     ws[u] := 0;
33 lclock := 0;
34 commit
```

of the variables of the STM algorithm. We use the notation `owner`[V] to denote that `owner` is an array of size V. All STM algorithms also consist of a program p_a which corresponds to the abort of the transaction.

TL2 Algorithm 3 shows four RML programs: p_r (read), p_w (write), p_e (end), and p_a (abort). The program p_a can be called from within p_r, p_w, and p_e. The global variables are `lock`[V], `version`[V], `g`[V], and `clk`. The local variables are `rs`[V], `ws`[V], `lver`[V], `lclock`, `c`, and `l`. The index variables are `u` and `v`. `self` denotes the thread number of the executing thread. The initial valuation of all variables is 0. The array `g`[V] of global variables corresponds to the addresses of the transactional variables V.

Algorithm 4 The DSTM algorithm in RML

```
01  program p_a :
02  u := 0;
03  while u < V do
04    u := u + 1;
05    l := cas(txr[v].tid, self, aborTx)
06  abort

01  program p_r :
02  p_w
03  rfin

01  program p_e :
02  u := 0;
03  while u < V do
04    u := u + 1;
05    l := cas(txr.tid[v], self, commTx)
06  commit

01  program p_w :
02  k := 0
03  l := txr[v];   t := l.tid
04  if t ≠ self then
05    while k ≠ l do
06      k := 0
07      m.tid := self
08      if t.status = 1 then
09        m.oldv := l.newv
10        m.newv := l.newv
11      if t.status = 2 then
12        m.oldv := l.oldv
13        m.newv := l.oldv
14      if t.status = 0 then
15        k := cas(t.status, 0, 2)
16        if k = 0 then
17          m.oldv := l.oldv
18          m.newv := l.oldv
19          k := cas(txr[v], l, m)
```

DSTM The global variables are txr[V]. The local variables are rs[V], v, k, l, m, t, and u. All variables are initialized to 0. DSTM consists of structures which can be implemented using a sequence of addresses. We use these structures and slightly modify the DSTM algorithm to avoid dynamic memory allocation. We express the programs p_r, p_w, and p_e of DSTM in RML in Algorithm 4.

McRT STM McRT STM [34] is an STM proposal from Intel. It significantly differs from the previous two STM algorithms we discussed, due to the fact that McRT STM is a direct update STM. McRT STM updates the global memory during the write command. The global variables are owner[V] and g[V]. The local variables are rs[V], l, m, u, and v. All variables are initialized to 0. The array g[V] of global variables corresponds to the addresses of the transactional variables V. McRT STM is presented in RML in Algorithm 5.

6 The FOIL tool

We developed a stateful explicit-state model checker, FOIL, that takes as input the RML description of an STM algorithm A, a memory model M, and a correctness property π, and checks whether A is correct with two threads and two variables for π under the memory model M. FOIL uses the RML semantics with respect to the memory model M to compute the state space of the STM algorithm A, and checks inclusion within the correctness property π. FOIL builds on the fly, the product of the transition system for A and the TM specification for π. In our case, we let the correctness criterion be opacity. If an STM algorithm A is not opaque for a memory model M, FOIL automatically inserts fences within the RML representation of A in order to make A opaque. FOIL succeeds if it is indeed possible to make A opaque solely with the use of fences. In this case, FOIL reports a possible set of missing fences. FOIL fails if inserting fences cannot make A opaque. In this case, FOIL produces a

Algorithm 5 The McRT-STM algorithm in RML

```
01  program p_a :
02  u := 0;
03  while u < V do
04    u := u + 1;
05    l := owner[u];
06    if l = self then
07      rollback[v];
08      owner[v] := 0;
09    rs[v] := 0
10  abort
```

```
01  program p_w :
02  l := owner[v];
03  if l ≠ self then
04    if l ≠ 0 then p_a
05    l := cas(owner[v], 0, self);
06    if l ≠ self then p_a
07    g[v] := l
```

```
01  program p_r :
02  if (rs[v] ≠ 1) then
03    l := owner[v];
04    if l ≠ self and l ≠ 0 then
05      if l < R then p_a
06      else
07        m := cas(owner[v], l, l + 1);
08        if m ≠ l + 1 then p_a
09        rs[v] := 1
10        l := g[v]
11        rfin
```

```
01  procedure p_e :
02  v := 0;
03  while v < V do
04    v := v + 1;
05    if rs[v] = 1 then
06      l := 0; m := 0
07      while (l − 1 ≠ m)
08        l := owner[v];
09        m := cas(owner[v], l, l − 1);
10      rs[v] := 0
11    else
12      if owner[v] = self then
13        owner[v] := 0
14  commit
```

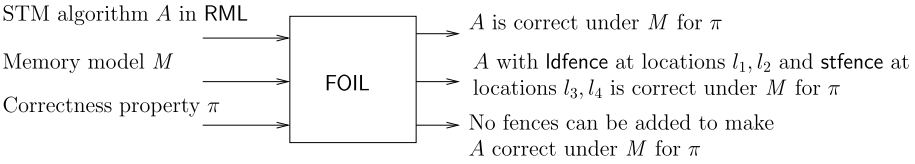

Fig. 5 Inputs and examples of possible outputs of FOIL

shortest counterexample to opacity under sequential consistency.[2] This is shown in Fig. 5. We used FOIL to check the opacity of DSTM, TL2, and McRT STM under different memory models.

6.1 Results

We first present the results we obtained from FOIL. Then, we provide some implementation details which make FOIL work.

[2]Note that if an STM algorithm A cannot be made opaque with fences under some memory model M, then A is not opaque even under sequential consistency.

Table 2 Time for checking the opacity of STM algorithms under sequential consistency on a 2.8 GHz PC with 2 GB RAM. The time is divided into time t_g needed to generate the language of the STM algorithm from the RML description, and time t_i needed to check inclusion within the property of opacity

STM algorithm A	Number of states	A is opaque?	t_g	t_i
TL2	1888674	Yes	581 s	1.1 s
DSTM	3060158	Yes	1327 s	2.3 s
McRT STM	479234	Yes	265 s	0.9 s

Table 3 Counterexamples generated for opacity, and the type and location of fences required to remove all counterexamples on different relaxed memory models. We list the statement number after which the fence has to be inserted in the RML program

STM	TSO	PSO	RMO
TL2	No fences	w_1, stfence: p_e, 26	w_1, stfence: p_e, 26
			w_3, ldfence: p_e, 17
			w_4, ldfence: p_r, 07
DSTM	No fences	No fences	No fences
McRT STM	No fences	w_2, stfence: p_a, 07	w_2, stfence: p_a, 07
Counterexamples			
$w_1 : (\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1), (\langle \text{store } v_1\rangle, t_2), (\langle \text{store } v_1\rangle, t_1)$			
$w_2 : (\langle \text{store } v_1\rangle, t_1), (\langle \text{load } v_2\rangle, t_2), (\langle \text{rfin}\rangle, t_2), (\langle \text{load } v_1\rangle, t_2), (\langle \text{rfin}\rangle, t_2), (\langle \text{rollback } v_1\rangle, t_1)$			
$w_3 : (\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1), (\langle \text{load } v_2\rangle, t_2), (\langle \text{rfin}\rangle, t_2), (\langle \text{store } v_1\rangle, t_2), (\langle \text{store } v_2\rangle, t_1)$			
$w_4 : (\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1), (\langle \text{store } v_1\rangle, t_2), (\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1)$			

Sequential consistency We first model check the STM algorithms for opacity on a sequentially consistent memory model. We find that all of DSTM, TL2, and McRT STM are opaque. The state space obtained for these STM algorithms is large as it covers every possible interleaving, where the level of atomicity is that of the hardware. Table 2 lists the number of states of different STM algorithms with the verification results under sequential consistency. The usefulness of FOIL is demonstrated by the size of the state spaces it can handle.

Relaxed memory models Next, we model check the STM algorithms on the following relaxed memory models: TSO, PSO, and RMO. We find that TL2 and McRT STM are not opaque for PSO and RMO. FOIL gives counterexamples to opacity. We let FOIL insert fences automatically until the STM algorithms are opaque under different memory models. Table 3 lists the number and location of fences inserted by FOIL to make the various STM algorithms opaque under various memory models. Note that the counterexamples shown in the table are projected to the loads, stores, and rollbacks of the transactional variables, and rfin instructions. We omit the original long counterexamples (containing for example, a sequence of loads and stores of locks and version numbers) for brevity.

Currently, STM designers use intuition to place fences, as lack of fences risks correctness, and too many fences hamper performance. As FOIL takes as input a memory model, it makes it easy to customize an STM implementation according to the relaxations allowed by the memory model. Although FOIL is not guaranteed to put the minimal number of fences, we found that FOIL indeed inserts the same fences as those in the official STM implementations.

Algorithm 6 Obtaining the transition system of an STM algorithm

generateTransitionSystem(A)

$frontier := \{q_{init}\}$
$Q := frontier$
while $frontier \neq \emptyset$
  pick and remove a state q from $frontier$
  $T := findnext(A, q)$
  $\delta := \delta \cup T$
  let Q' be the set of destination states in T
  add the set $Q \setminus Q'$ of states to $frontier$
  $Q := Q \cup Q'$
output δ

6.2 Analysis

We note that reordering a store followed by a load, and reading own write early (due to store buffers) does not create a problem in the STMs we have studied. This is evident from the fact that all STMs are correct under the TSO memory model without any fences. On the other hand, relaxing the order of stores or loads can be disastrous for the correctness of an STM. This is because most STMs use version numbers or locks to control access. For example, a reading thread first checks that the variable is unlocked and then reads the variable. A writing thread first updates the variable and then unlocks it. Reversing the order of writes or reads renders the STM incorrect.

7 Implementation details

We implemented FOIL in OCaml. FOIL supports two modes: generating the state space of an STM algorithm and finding a counterexample history on-the-fly. We found it important to allow both modes in FOIL due to the following reason. The state spaces of the STM algorithms are very large. Checking for a counterexample on the fly requires to build the product automaton. The state space of the product automaton could be as large as the product of the state spaces of the STM algorithm and the TM specification. The size of the transition system of the STM algorithms is, on average, a million states under sequential consistency. The size is much bigger under highly relaxed memory models. The size of the deterministic TM specification is around 46,000 states. We found it impossible, with modest computing resources, to construct the whole product automaton or the transition system of an STM algorithm under a highly relaxed memory model using FOIL. However, FOIL could construct the transition system of the STM algorithms under sequential consistency. We then use a lightweight language inclusion tool to check whether the language of the STM algorithm is included in the language of the TM specification.

7.1 Generating the state space

We obtain the transition relation of an STM algorithm using Algorithm 6. Intuitively, the algorithm finds all states reachable from the initial state, and outputs the transition system. How FOIL handles relaxed memory models, is hidden beneath the *findnext* procedure. The *findnext* procedure takes as input an STM algorithm and a state q of the STM algorithm, and gives all transitions in the STM algorithm from the state q.

Algorithm 7 Obtaining a counterexample to opacity on-the-fly

findCounterexample$(A, Spec)$

$frontier := \{(q_{init}, p_{init})\}$
$path(q_{init}, p_{init}) := \varepsilon$
$Q := frontier$
while $frontier \neq \emptyset$
  pick and remove a state (q, p) from $frontier$
  $T := findnext(A, q)$
  for each transition $(q, op, q') \in T$ **do**
    if $op \in \hat{Op}$ **then**
      if there exists a state $p_1 \in P$ such that $(p, op, p_1) \in \delta_p$ **then**
        $p' := p_1$ such that $(p, op, p_1) \in \delta_p$
      else
        report counterexample $path(q, p) \cdot op$
    else $p' := p$
    if $(q', p') \notin Q$ **then**
      add (q', p') to Q
      add (q', p') to $frontier$
      $path(q', p') = path(q, p) \cdot op$
report no counterexample found

7.2 Finding a counterexample

Although the state space generation mode of FOIL gives the transition system for the STM algorithms under sequential consistency, the mode cannot produce the transition system for the STM algorithms under relaxed memory models. This is because the state space of an STM algorithm under a relaxed memory model can be too large to explore with modest computation speed and memory. However, many histories produced under these memory models may not even satisfy opacity. To handle this situation, we use on-the-fly verification. FOIL maintains the product automaton of the transition system of the STM algorithm and the TM specification, and tries to find counterexamples early. FOIL uses Algorithm 7 to construct the product automaton and find a counterexample to opacity. We represent the deterministic TM specification for opacity as $Spec = \langle P, p_{init}, \delta_p \rangle$.

We find this procedure highly successful, because counterexamples of opacity tend to be short. On observing a counterexample, FOIL suggests the location of a fence which might make the STM algorithm correct under the given relaxed memory model. On inserting the fence, the number of interleavings decreases, and thus the size of the state space decreases. We run FOIL in this mode until it takes a few minutes to find a counterexample. After that, we run FOIL in the state space generation mode. Once we obtain the state space of the STM algorithm, we can use our lightweight language inclusion tool to check whether the language of the STM algorithm is included in the specification. The interesting part is how the two modes help each other. It is not possible to reach our goal with either of the modes. The state space generation mode fails to generate the state space of an STM algorithm under a highly relaxed memory model due to the large number of interleavings. The counterexample finder mode fails to finish due to the large size of the product automaton even under sequential consistency.

7.3 Counterexample analysis

Our tool FOIL automatically inserts fences. However, FOIL does not ensure that the number of fences it inserts is minimal. We describe how FOIL chooses the place where the fence needs to be inserted.

Recall that when FOIL encounters a statement s with a memory instruction *inst* of thread t, FOIL adds the instruction in the queue $d = s_0 \dots s_n$ of deferred statements of thread t according to the given memory model. If FOIL inserts the memory instruction in the middle of the queue to obtain $d' = s_0 \dots s_i \cdot s \cdot s_{i+1} \dots s_n$, FOIL tags the statement s in the queue with the string $s_n \cdot s_{n-1} \dots s_{i+1}$. When FOIL reports a counterexample, we search for the last statement with a reordering tag in the counterexample. Let the tag be $s_1 \dots s_k$. We attribute the error to the reordering allowed by s_k. So, we insert a fence after the statement s_k. The inserted fence is a store (resp. load) fence if s_k is a store (resp. load) instruction.

8 Extending the verification results

We now extend the verification results for two threads and two variables to an arbitrary number of threads and variables.

8.1 Structural properties of STM algorithms

In earlier work [19], we presented a set of structural properties for STMs on a high level alphabet. These properties are hard to directly prove with hardware level atomicity. So, we present sufficient conditions for these structural properties to hold. For the sake of completeness, we present the structural properties again in this paper.

To reduce the verification problem of STM algorithms to a finite number of threads and variables, we reason about STM algorithms in terms of the following properties. We assume a memory model M.

- An STM algorithm A is *abort isolated* if for every history $h \in L(A, M)$, for every aborted transaction x in h, if an instruction *inst* of x changes the value of a global variable g and a transaction y observes the value of g before x aborts, then y aborts in the step of observing g.
- An STM algorithm is *pending isolated* if for every history h, for every pending transaction x in h, if an instruction *inst* of x changes the value of a global variable g and a transaction y observes the value of g before x finishes, then y aborts.
  These two properties restrict the aborting and pending transactions in an STM. These properties require that if an aborting or a pending transaction changes the global state, then that change is not visible to committing transactions. We shall later use these properties to remove the aborting and pending transactions from a history.
- A STM algorithm is *conflict commutative* if for every history h where the execution of a transactional command c_1 by thread t overlaps with the execution of a transactional command c_2 by thread u, if c_1 consists of an instructions $inst_1$ and $inst_3$, and c_2 consists of instructions $inst_2$ and $inst_4$ such that $inst_1$ occurs before $inst_2$ in h and they conflict, and $h = h_1 inst_4 inst_3 h_2$, then $h' = h_1 inst_3 inst_4 h_2$ is also a history in the language of the STM algorithm.

We now present four structural properties of STM algorithms. We then use these properties to prove the reduction theorem for opacity. We use the above definitions to show that the

STM algorithms, DSTM and TL2, satisfy these structural properties. Note that the properties are sufficient (and not necessary) conditions for the reduction theorem to hold. Let Γ be a transactional memory and let A be the corresponding STM algorithm. Let h be a history in Γ.

P1 Transaction projection We define the *transaction projection* of h on $X' \subseteq X$ as the subsequence of h that contains every statement of all transactions in X'. The property P1 states that the transaction projection of h on X' is in Γ, where X' contains all committed transactions and no aborted transactions.

We should note that, however, we cannot project away a subset of the aborted transactions. This is because removing an aborted transaction may allow another aborted transaction to commit.

Lemma 1 *If an STM algorithm A is abort isolated and pending isolated, then the STM Γ satisfies transaction projection.*

Proof Consider an arbitrary history $h \in L(A, M)$. We can divide the history h into subsequences $h_1 \dots h_n$, where for all i, all statements in h_i are committing, aborting, or pending. As the STM algorithm is abort isolated, we can remove the subsequences from $h_1 \dots h_n$ which are aborting. Moreover, as the STM algorithm is pending isolated, we can remove a subset of the subsequences which are pending. Hence, we get a new history h' such that all statements in h' belong to committed or pending transactions, and $h' \in L(A, M)$. □

P2 Thread renaming For non-overlapping transactions, the STM is oblivious to the identity of the thread executing the transaction. The property P2 states that if (i) h has no aborting transactions, and (ii) there exist two threads u and t such that for all committing transactions x of u and y of v in the history h, either $x <_w y$ or $y <_w x$, then the history h' obtained by renaming all transactions of thread u to be from thread t is in Γ.

P3 Variable projection If a transaction can commit, then removing all statements that involve some particular variables does not cause the transaction to abort. We define the *variable projection* of h on $V' \subseteq V$ as the subsequence of h that contains all commit and abort statements, and all read and write statements to variables in V'. The property P3 states that if h has no aborting transactions, then for all $V' \subseteq V$, the variable projection of h on V' is in Γ. An STM satisfies variable projection if reading or writing a variable does not remove a conflict on other variables.

P4 Monotonicity The most important property which allows to reduce the verification property is the monotonicity in STM. Monotonicity states that if a history is allowed by the STM, then more sequential forms of the history are also allowed. Formally, let $F \subseteq \hat{Op}^*$ be the set of opaque histories with all committed and exactly one unfinished transaction. We define a function $seq : F \to 2^F$ such that if $h_2 \in seq(h_1)$, then h_2 is sequential and strictly equivalent to h_1. The monotonicity property for opacity states that if $h = h' \cdot op$, where $h' \in F$, and op is not an abort, and op is an operation of the unfinished transaction in h', then for every history $h_2 \in seq(h')$, the history $h_2 \cdot op$ is a finite prefix of a history in Γ.

Lemma 2 *If the STM algorithm A is conflict commutative, then the STM Γ is monotonic.*

Proof Consider a history $h = h' \cdot op$ produced by the STM algorithm. We want to prove that if h' is opaque and h' consists of all committed and exactly one unfinished transaction, then h' can be sequentialized. We consider the history h'. For every pair of conflicting operations, we use conflict commutativity to sequentialize the corresponding commands. As h' is opaque, sequentializing the commands gives a sequential history such that the state of the STM algorithm after the sequential version of h' is equivalent to the state of the STM algorithm after h'. Thus, $h' \cdot op$ is produced by the STM algorithm. Thus, Γ is monotonic. □

The structural properties lead to the reduction theorem [19] which states that if an STM algorithm satisfies opacity for two threads and two variables, and satisfies the properties P1–P4, then the STM algorithm satisfies opacity for an arbitrary number of threads and variables.

8.2 Proving structural properties for STMs

We cannot prove that the STM algorithm McRT STM is abort isolated. This is because McRT STM is a direct update STM, which writes to memory during the transaction. Thus, a state change by a thread in the STM algorithm can be observed by other threads.

We note that for DSTM and TL2, the programs p_r, p_w, and p_e and the initial state z^{init} do not distinguish between the threads. Thus, the STM algorithms we consider satisfy the thread renaming P2 property. DSTM and TL2 satisfy P3 as they track every variable accessed by every thread independently.

8.2.1 DSTM

DSTM relies on a notion of ownership. If a transaction wants to read or write a variable, it first atomically sets the transaction record of the variable to itself. If another transaction wants to access the same variable, it first sets the status of the owner transaction to aborted, and reads the old value of the variable.

Abort and pending isolation An aborting or pending transaction in DSTM changes the global valuation by changing the status of other transactions to aborted. All other changes are not observable to committed transactions.

Conflict commutative DSTM uses encounter-time ownership for both reads and writes. Thus, if a transaction loads a variable before another transaction stores to the same variable, then all instructions of the *txrd* command can appear before all instructions of the *txwr* command.

8.2.2 TL2

Abort and pending isolation An aborting transaction x in TL2 can hold a lock and change the value of the global timestamps. If another transaction y observes that x holds the lock for some variable, y aborts. Similarly, if y observes an increment of the global timestamp by x during a read, y aborts. A similar argument holds for pending isolation.

Conflict commutative To prove that TL2 is conflict commutative at hardware level atomicity, we need to consider the relaxed memory model. It turns out that the TL2 algorithm (shown in Algorithm 3) is not conflict-commutative under some relaxed memory models. This is because if a *txend* command overlaps with a *txrd* command, such that the store instruction in the *txend* command appears before the load instruction in the *txrd* command, it may not be possible to move all instructions corresponding to the *txrd* command after all instructions of the *txend* command. This problem goes away if we insert some fences in the RML program of the TL2 algorithm. We observe that once FOIL inserts the required fences, TL2 is indeed conflict commutative. Consider a *txrd* command overlapping with a conflicting *txend* command. If the transactional variable is loaded in *txrd* before the variable is stored in *txend*, then all instructions of the *txrd* command can be moved above all instructions of the *txend* command (otherwise the reading transaction must abort). Similarly, if the *txend* commands of two transactions x and y overlap, if x stores to the transactional variable before y, then all instructions in the *txend* command of x can be moved above all instructions of the *txend* command of y.

9 Related work

We present a brief overview of the related work in the direction of relaxed memory models and verification tools for relaxed memory models.

9.1 Formalisms for relaxed memory models

Adve et al. [1] provide a detailed description of hardware relaxed memory models. Language level memory models have been developed for Java [29] and C++ [3]. Various formalisms for memory models have been proposed in the literature [4, 7, 21, 35, 36]. Most of these formalisms provide an axiomatic definition of memory models. Architectural manuals [39, 42] also describe memory models in an axiomatic style. Operational semantics of relaxed memory models were developed by Petri et al. [4]. Their formalism captures write-to-read/write reordering used in memory models like TSO and PSO. Moreover, the formalism allows thread creation. However, it cannot express read-to-read/write relaxations found in memory models like RMO.

Verification based on axiomatic memory model specifications relies on constraint solving (like SAT solvers) to validate execution traces. As our tool is based on explicit state model checking, we find it more intuitive to define an operational semantics of relaxed memory models. Our operational semantics handles write-to-read/write and read-to-read/write relaxations, but cannot handle thread creation.

9.2 Verification tools

Model checkers like Zing [2], SPIN [25], KISS [33], and CHESS [30, 32] are developed for verification of concurrent programs. These tools are built to detect races in concurrent programs. However, STM algorithms, by design, often consist of benign races. Moreover, these tools assume a sequentially consistent memory model, and miss out a whole range of interleavings that arise due to the reorderings of the instructions of a thread allowed by a relaxed memory model. Verification of concurrent data types has been attempted with theorem proving [9, 41]. These methods require a manual tedious proof construction, and assume sequential consistency.

Dynamic tools [13, 15–17] for verifying atomicity and race freedom in concurrent programs have also been built. Manovit et al. [28] used testing to find errors in STM implementations. Elmas et al. [12] have built tools for runtime verification of concurrent data types. The motivation for dynamic tools is to check the correctness of a computation at runtime, and throw an exception in case of error. Dynamic tools can find errors in an STM algorithm only when the STM algorithm is used to execute a transactional program. Dynamic tools cannot be used to establish the correctness of an STM algorithm, that is, to check whether the STM algorithm is correct for all programs. However, as static analysis is tricky for real iSTM implementations, we believe that dynamic tools can be useful to check correctness properties of STM implementations at runtime. Instead of the data race specification, the specification of opacity constructed in this thesis could be used.

Burckhardt et al. [5, 6] developed CheckFence, a static verification tool for concurrent C programs under relaxed memory models. The tool requires as input a bounded test program (a finite sequence of operations) for a concurrent data type and uses a SAT solver to check the consistency and report if any fences are required. However, CheckFence cannot automatically introduce fences. We use the structural properties of STM, which allow us to consider a maximal program on two threads and two variables in order to generalize the result to all programs with any number of threads and variables. We model the correctness problem as a relation between transition systems. Moreover, our tool, FOIL, automatically inserts fences. Padua et al. [14, 27] developed mechanisms to ensure sequential consistency under relaxed memory models. However, conservatively putting fences into STM implementations to guarantee sequential consistency would badly hurt STM performance. STM programmers put fences only where necessary. Gopalakrishnan et al. [18] developed a verification tool for checking memory orderings for small programs.

9.3 STM Verification

Recent work has addressed verification in the context of transactional memories. Tasiran [40] verified the correctness of the Bartok STM. The author manually proves the correctness of the Bartok STM algorithm, and uses assertions in the Bartok STM implementation to ensure that the implementation refines the algorithm. This work is orthogonal to ours, as we focus on automated techniques to prove the correctness of TM algorithms. Cohen et al. [8] model checked STM applied to programs with a small number of threads and variables, against the strong correctness property of Scott [37]. Further, they studied safety properties in situations where transactional code has to interact with non-transactional accesses. Guerraoui et al. [19, 20] presented specifications for strict serializability and opacity in STM algorithms and model checked various STMs. All these verification techniques in STMs assumed sequentially consistent execution and the atomicity of STM operations like read, write, and commit.

10 Conclusion

We address the verification issues related to STM implementations on real multiprocessors. First of all, we presented a formalism to express STMs and their correctness properties at the hardware level of atomicity under relaxed memory models. We illustrated our formalism by specifying common STMs such as DSTM, TL2, and McRT STM; memory models such as total store order (TSO), partial store order (PSO), and relaxed memory order (RMO); and correctness criteria such as opacity. We then presented a tool, FOIL, which automatically

checks the correctness of STMs under fine-grained hardware atomicity and relaxed memory models. FOIL can automatically insert load and store fences where necessary in the STM algorithm description, in order to make the STMs correct under various relaxed memory models. We plan to extend our work to more complicated software memory models, such as Java [29], which further relax the order of memory instructions.

References

1. Adve SV, Gharachorloo K (1996) Shared memory consistency models: A tutorial. IEEE Comput 66–76
2. Andrews T, Qadeer S, Rajamani SK, Rehof J, Xie Y (2004) Zing: A model checker for concurrent software. In: International conference on computer aided verification. Springer, Berlin, pp 484–487
3. Boehm HJ, Adve SV (2008) Foundations of the C++ concurrency memory model. In: ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 68–78
4. Boudol G, Petri G (2009) Relaxed memory models: An operational approach. In: ACM SIGPLAN symposium on principles of programming languages, pp 392–403
5. Burckhardt S, Alur R, Martin MMK (2006) Bounded model checking of concurrent data types on relaxed memory models: A case study. In: International conference on computer aided verification. Springer, Berlin, pp 489–502
6. Burckhardt S, Alur R, Martin MMK (2007) CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 12–21
7. Burckhardt S, Musuvathi M, Singh V (2008) Verifying compiler transformations for concurrent programs. Technical Report MSR-TR-2008-171, Microsoft Research
8. Cohen A, Pnueli A, Zuck LD (2008) Mechanical verification of transactional memories with non-transactional memory accesses. In: International conference on computer aided verification. Springer, Berlin, pp 121–134
9. Colvin R, Groves L, Luchangco V, Moir M (2006) Formal verification of a lazy concurrent list-based set algorithm. In: International conference on computer aided verification. Springer, Berlin, pp 475–488
10. De Wulf M, Doyen L, Henzinger TA, Raskin J-F (2006) Antichains: A new algorithm for checking universality of finite automata. In: International conference on computer aided verification. Springer, Berlin, pp 17–30
11. Dice D, Shalev O, Shavit N (2006) Transactional locking II. In: International symposium on distributed computing. Springer, Berlin, pp 194–208
12. Elmas T, Tasiran S, Qadeer S (2005) VYRD: Verifying concurrent programs by runtime refinement-violation detection. In: ACM SIGPLAN conference on programming language design and implementation, pp 27–37
13. Elmas T, Qadeer S, Tasiran S (2007) Goldilocks: A race and transaction-aware Java runtime. In: ACM SIGPLAN conference on programming language design and implementation, pp 245–255
14. Fang X, Lee J, Midkiff SP (2003) Automatic fence insertion for shared memory multiprocessing. In: International conference on supercomputing, pp 285–294
15. Flanagan C, Freund SN (2004) Atomizer: A dynamic atomicity checker for multithreaded programs. In: ACM SIGPLAN symposium on principles of programming languages, pp 256–267
16. Flanagan C, Freund SN (2009) FastTrack: Efficient and precise dynamic race detection. In: ACM SIGPLAN conference on programming language design and implementation, pp 121–133
17. Flanagan C, Freund SN, Yi J (2008) Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In: ACM SIGPLAN conference on programming language design and implementation, pp 293–303
18. Gopalakrishnan G, Yang Y, Sivaraj H (2004) QB or Not QB: An efficient execution verification tool for memory orderings. In: International conference on computer aided verification. Springer, Berlin, pp 401–413
19. Guerraoui R, Henzinger TA, Jobstmann B, Singh V (2008) Model checking transactional memories. In: ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 372–382
20. Guerraoui R, Henzinger TA, Singh V (2008) Nondeterminism and completeness in model checking transactional memories. In: International conference on concurrency theory. Springer, Berlin, pp 21–35
21. Guerraoui R, Henzinger TA, Singh V (2009) Software transactional memory on relaxed memory models. In: International conference on computer aided verification. Springer, Berlin, pp 321–336

22. Guerraoui R, Kapałka M (2008) On the correctness of transactional memory. In: ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, New York, pp 175–184
23. Herlihy M, Moss JEB (1993) Transactional memory: Architectural support for lock-free data structures. In: International symposium on computer architecture. ACM, New York, pp 289–300
24. Herlihy M, Luchangco V, Moir M, Scherer WN (2003) Software transactional memory for dynamic-sized data structures. In: ACM SIGACT-SIGOPS symposium on principles of distributed computing. ACM, New York, pp 92–101
25. Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 279–295
26. Lamport L (1979) How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans Comput 690–691
27. Lee J, Padua DA (2001) Hiding relaxed memory consistency with a compiler. IEEE Trans Comput 824–833
28. Manovit C, Hangal S, Chafi H, McDonald A, Kozyrakis C, Olukotun K (2006) Testing implementations of transactional memory. In: International conference on parallel architectures and compilation techniques, pp 134–143
29. Manson J, Pugh W, Adve SV (2005) The Java memory model. In: ACM SIGPLAN symposium on principles of programming languages. ACM, New York, pp 378–391
30. Musuvathi M, Qadeer S, Ball T, Basler G, Nainar PA, Neamtiu I (2008) Finding and reproducing heisenbugs in concurrent programs. In: USENIX symposium on operating systems design and implementation, pp 267–280
31. Papadimitriou CH (1979) The serializability of concurrent database updates. J ACM 26(4)
32. Qadeer S, Rehof J (2005) Context-bounded model checking of concurrent software. In: International conference on tools and algorithms for the construction and analysis of systems, pp 93–107
33. Qadeer S, Wu D (2004) KISS: Keep it simple and sequential. In: ACM SIGPLAN conference on programming language design and implementation, pp 14–24
34. Saha B, Adl-Tabatabai A, Hudson RL, Minh CC, Hertzberg B (2006) McRT-STM: A high performance software transactional memory system for a multi-core runtime. In: ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, New York, pp 187–197
35. Saraswat VA, Jagadeesan R, Michael M, von Praun C (2007) A theory of memory models. In: ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, New York, pp 161–172
36. Sarkar S, Sewell P, Zappa Nardelli F, Owens S, Ridge T, Braibant T, Myreen MO, Alglave J (2009) The semantics of x86-CC multiprocessor machine code. In: ACM SIGPLAN symposium on principles of programming languages, pp 379–391
37. Scott ML (2006) Sequential specification of transactional memory semantics. In: ACM SIGPLAN workshop on transactional computing
38. Shavit N, Touitou D (1995) Software transactional memory. In: ACM SIGACT-SIGOPS symposium on principles of distributed computing. ACM, New York, pp 204–213
39. Sites RL (ed) (2002) Alpha architecture reference manual. Digital Press, Newton
40. Tasiran S (2008) A compositional method for verifying software transactional memory implementations. Technical Report MSR-TR-2008-56, Microsoft Research
41. Vafeiadis V, Herlihy M, Hoare T, Shapiro M (2006) Proving correctness of highly-concurrent linearisable objects. In: ACM SIGPLAN symposium on principles and practice of parallel programming, pp 129–136
42. Weaver D, Germond T (eds) (1994) The SPARC architecture manual (version 9). Prentice-Hall Inc, Englewood Cliffs