Model Checking Transactional Memories *

Rachid Guerraoui Thomas A. Henzinger Barbara Jobstmann Vasu Singh

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{rachid.guerraoui,tah,barbara.jobstmann,vasu.singh}@epfl.ch

Abstract

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

Our main contribution lies in the structure of the proofs, which are largely automated and not restricted to the STMs mentioned above. In a first step we show that every STM that enjoys certain structural properties either violates a safety or liveness requirement on some program with two threads and two shared variables, or satisfies the requirement on all programs. In the second step we use a model checker to prove the requirement for the STM applied to a most general program with two threads and two variables. In the safety case, the model checker constructs a simulation relation between two carefully constructed finite-state transition systems, one representing the given STM applied to a most general program, and the other representing a most liberal safe STM applied to the same program. In the liveness case, the model checker analyzes fairness conditions on the given STM transition system.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent Programming; D.2.4 [*Software engineering*]: Software/Program Verification

General Terms Languages, Verification

Keywords Transactional memories, Model checking

1. Introduction

With the advent of multi-core processors, there is a new urgency for concurrent programming models that give the programmer the illusion of sequentiality and the compiler maximal flexibility. A model that has enjoyed particular recent success is software transactional memory (STM), which allows the programmer to think in coarse-grained code blocks that appear to be executed atomically and, at the same time, minimally constrains the compiler. Inspired by how databases manage concurrency, transactional memory was first introduced by Herlihy and Moss [HM93] in multi-processor design. Later Shavit and Touitou [ST95] introduced STM, a software-based variant of the concept, which enables a new way of looking at concurrent programming. An extensive overview of STM can be found in [LR07]. In this paper, we consider the following STM algorithms: two-phase locking, DSTM [HLMS03], TL2 [DSS06], and optimistic concurrency control [KR81].

Precisely because STM algorithms encapsulate the difficulty of handling concurrency, the potential of subtle errors is enormous. This makes STM a ripe and important proving ground for formal verification. While there have been initial steps in this direction [COP+07], the challenge remains daunting for several reasons.

First, there is no generally agreed upon formal notion of correctness for STM. Scott [Sco06] was the first to provide a formal semantics for STM. However, his weakest correctness criterion requires the order of commits to be preserved. Thus, the popular STM algorithm TL2 [DSS06], which does not preserve the order of commits, falls outside the semantic classification by Scott. Guerraoui and Kapalka [GK08] discussed various alternatives to precisely capture the safety aspect of STM and highlighted the subtle differences with database transactions.

Second, while model checking is the verification technique that is best equipped to find concurrency bugs, model checking is severely handicapped by several sources of unbounded state in STM: memory size, thread count, and transaction length cannot be bounded, and neither can the delay until a transaction commits, nor the number of times that a transaction aborts. As with relaxed memory models, special care is needed in formulating a verification problem that is both relevant and solvable, as some problems about sequentializing concurrent systems are undecidable [AMP00].

Third, the specification of an STM universally quantifies over all possible application programs, requiring the desired safety and liveness conditions *for all* programs that are executed on the STM. In this sense, STM verification resembles the problem of checking that a processor implements an instruction set architecture, where the executed programs are also universally quantified. In both cases, the key is to define (and check) a suitable implementation relation [BD94]. While in processor verification, the implementation relation needs to handle pipelines and out-of-order execution, in STM, we need to handle aborted transactions.

We present in this paper a new technique for verifying STM safety and liveness properties. Our technique addresses the three issues above as follows.

First, the safety requirements we consider are *strict serializability* [Pap79] and *opacity* [GK08]. (We consider a single-version read/write restriction of the general notion of opacity.) Strict serializability preserves the order of conflicting operations between transactions, and the order of non-overlapping transactions. Opacity ensures, in addition, that aborting transactions do not see an inconsistent state of the memory, which can be disastrous in STMs

* This research was supported by the Swiss National Science Foundation.

PLDI'08 June 7–13, 2008, Tucson, Arizona, USA.

(due to infinite loops, or exceptions). We study opacity, because it provides the programmer with the full sequentiality illusion and is satisfied by most STM protocols that claim that illusion [LR07]. Strict serializability is considered here for pedagogical reasons, as it is intuitive and captures the main technical difficulties behind verifying opacity. The liveness requirements we consider are the standard notions of *obstruction freedom* [HLM03], *livelock freedom* [AKH03], and *wait freedom* [Her91].

Second, we exploit the structural symmetries that are inherent in STM algorithms to reduce the verification of unbounded STM state spaces to a problem that involves only a small number of threads and shared variables. Specifically, we show that every STM that enjoys certain structural properties either violates any of the considered safety and liveness requirements on some program with two threads and two shared variables, or satisfies the requirement on all programs. The structural properties, which expect all threads to be treated equally, are fulfilled by most transactional algorithms, including for instance, two-phase locking, DSTM, TL2, and optimistic concurrency control. Similar techniques for reducing unbounded instances of model-checking tasks to small, characteristic instances have been used for verifying protocols with an unbounded number of identical processes [BCG89] and cache-coherence protocols [HQR99].

Third, and perhaps most importantly, we define two finite-state transition systems that generate exactly the strictly serializable (resp. opaque) executions of programs with two threads and two shared variables. These transition systems can be viewed as most liberal *reference STM algorithms* guaranteeing strict serializability (resp. opacity). To our knowledge, the transition systems presented in this paper provide the first finite-state representation of the language of strictly serializable (resp. opaque) executions for transactions that may abort. The finite size of the transition systems is achieved by a careful choice of state, which encompasses for every thread a set of read variables (at most two), a set of written variables (at most two), a set of variables not allowed to be read (at most two), a set of variables not allowed to be written (at most two), and a set of threads with overlapping, preceding transactions (at most 1). We show that an STM algorithm is strictly serializable (resp. opaque) iff for a specific, most general program with two threads and two variables, all executions are permitted by the reference STM algorithm. Then, instead of checking language containment between a given STM algorithm and the reference algorithm, we check for the existence of a simulation relation between both transition systems [Mil71]. The existence of a simulation relation is a commonly used, efficient sufficient condition for language containment.

Putting all steps together, we reduce the problem of verifying the safety of an STM algorithm, which is unbounded in many dimensions (memory size, thread count, transaction delay, etc.), to a simulation check between two finite-state systems. For two-phase locking, DSTM, TL2, and optimistic concurrency control, we obtain transition systems with up to 12,000 states, and the reference transition systems have about 12,500 states. We implemented a simulation checker that automatically verifies strict serializability for optimistic concurrency control and opacity for two-phase locking, DSTM, and TL2 in less than 30 minutes. It should be noted that the methodology is applicable to any other STM algorithms that satisfy the structural properties. Our simulation checker finds that correctness is not self-evident in many STM algorithms. For example, we found an ambiguity in ordering of two particular operations in the published TL2 algorithm [DSS06]. One of the orderings makes TL2 unsafe. In this case, the simulation check provides as counterexample an execution that is not strictly serializable (and thus not opaque). We therefore expect our verification tool to be useful to STM designers when they develop or modify STM algorithms.

On the liveness side, we prove again a structural reduction theorem to check the desired liveness requirement on the finite-state transition system that results from a given STM algorithm applied to a most general program with two threads and one variable. We built a model checking tool to verify the different liveness properties. In the case of obstruction freedom, this amounts to checking a Streett condition. The check goes through for DSTM. For two-phase locking, TL2, and optimistic concurrency control, the model checker automatically generates counterexamples to obstruction freedom, as it does for DSTM and livelock freedom.

2. Safety in transactional memories

We introduce a few notions about transactions, and then formalize the correctness of transactional memories.

Let V be a set $\{1,\dots,k\}$ of k *variables*, and let $C = \{\mathsf{commit}\} \cup (\{\mathsf{read},\mathsf{write}\} \times V)$ be the set of *commands* on the variables V. Also, let $\hat{C} = C \cup \{\mathsf{abort}\}$. Let $T = \{1,\dots,n\}$ be a set of n *threads*. Let $\hat{S} = \hat{C} \times T$ be the set of *statements*. Also, let $S = C \times T$. A word $w \in \hat{S}^*$ is a finite sequence of statements. Given a word $w \in \hat{S}^*$, we define the *thread projection* $w|_t$ of w on thread $t \in T$ as the subsequence of w consisting of all statements s in w such that $s \in \hat{C} \times \{t\}$. Given a thread projection $w|_t = s_0 \dots s_m$ of a word w on thread t, a statement s_i is *finishing in* $w|_t$ if it is a commit or an abort. A statement s_i is *initiating in* $w|_t$ if it is the first statement in $w|_t$, or the previous statement s_{i-1} is a finishing statement.

Given a thread projection $w|_t$ of a word w on thread t, a consecutive subsequence $x = s_0 \dots s_m$ of $w|_t$ is a *transaction* of thread t in w if (i) s_0 is initiating in $w|_t$, and (ii) s_m is either finishing in $w|_t$, or s_m is the last statement in $w|_t$, and (iii) no other statement in x is finishing in $w|_t$. The transaction x is *committing* in w if s_m is a commit. The transaction x is *aborting* in w if s_m is an abort. Otherwise, the transaction x is *unfinished* in w. Given a word w and two transactions x and y in w (possibly of different threads), we say that x *precedes* y in w, written as $x <_w y$, if the last statement of x occurs before the first statement of y in w. A word w is *sequential* if for every pair x, y of transactions in w, either $x <_w y$ or $y <_w x$. We define a function $com : \hat{S}^* \to S^*$ such that for all words $w \in \hat{S}^*$, the word $com(w)$ is the subsequence of w that consists of every statement in w that is part of a committing transaction.

A transaction x of a thread t *writes* to a variable v if x contains a statement $((\mathsf{write}, v), t)$. A statement $s = ((\mathsf{read}, v), t)$ in x is a *global read* of a variable v if there is no statement $((\mathsf{write}, v), t)$ before s in the transaction x. A transaction x of a thread t *globally reads* a variable v if there exists a global read of variable v in transaction x. A word w is *transaction equivalent* to a word w' if for every thread $t \in T$, we have $w|_t = w'|_t$. Note that two transaction equivalent words have the same order of commands for all threads.

2.1 Safety criteria

Conflict serializability [EGLT76] is a commonly used correctness criterion for concurrent systems and, in particular, for transactional systems. Conflict serializability allows us to omit the values of read and write commands, since the consistency of the values follows from preserving the order of conflicts. In the context of transactional memories, a stronger property, called strict serializability, is considered. Strict serializability preserves the order of non-overlapping transactions too. We note that strict serializability does not state any restrictions on the operations of the aborting transactions. In the scope of STMs, an even stronger notion of correctness, referred to as *opacity*, has been suggested [HLMS03, GK08] to avoid unexpected side effects, like infinite loops, or array bound

violations. Opacity requires that a word be strictly serializable, and that even aborting transactions do not read inconsistent values.

Now, we formalize these correctness criteria. We start with the notion of a conflict. Transactional memories use direct update semantics (every transaction modifies the shared variables in place and restores them upon abort), or deferred update semantics (every transaction modifies a local copy, and changes the shared copy upon a commit). We choose to define conflicts under the deferred update semantics. A statement s_1 of transaction x and a statement s_2 of transaction y (where x is different from y) *conflict* in a word w if (i) s_1 is a global read of some variable v, and s_2 is a commit, and y writes to v, or (ii) s_1 and s_2 are both commits, and x and y write to the same variable v. A word $w = s_0 \dots s_m$ is *conflict equivalent* to a word w' if (i) w is transaction equivalent to w', and (ii) for every pair s_i, s_j of statements in w, if s_i and s_j conflict and $i < j$, then s_i occurs before s_j in w'. Note that transaction equivalence ensures that conflict equivalence is a symmetric relation, since w' is a permutation of w.

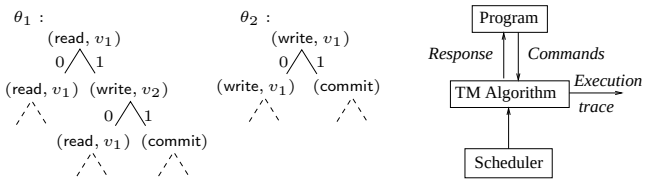
A word $w = s_0 \dots s_m$ is *strictly equivalent* to a word w' if (i) w is conflict equivalent to w', and (ii) for every pair x, y of transactions in w, where x is a committing or an aborting transaction, if $x <_w y$, then it is not the case that $y <_{w'} x$. A word $w \in \hat{S}^*$ is *strictly serializable* if there exists a sequential word w' such that w' is strictly equivalent to $com(w)$. Furthermore, a word w is *opaque* if there exists a sequential word w' such that w' is strictly equivalent to w. We note that given a word w, if w is opaque, then w is strictly serializable. An infinite word $w \in \hat{S}^\omega$ is *strictly serializable* (resp. *opaque*) if every finite prefix of w is strictly serializable (resp. opaque).

Example. Consider a word $w = ((\mathsf{read}, v_1), t_1), ((\mathsf{write}, v_1), t_2), ((\mathsf{write}, v_2), t_2), (\mathsf{commit}, t_2), ((\mathsf{read}, v_2), t_1), (\mathsf{abort}, t_1)$. w has two transactions: (i) an aborting transaction of t_1, and (ii) a committing transaction of t_2. The following pairs of statements conflict: $(((\mathsf{read}, v_1), t_1), (\mathsf{commit}, t_2))$ and $(((\mathsf{read}, v_2), t_1), (\mathsf{commit}, t_2))$. The word w is strictly serializable because $com(w) = ((\mathsf{write}, v_1), t_2), ((\mathsf{write}, v_2), t_2), (\mathsf{commit}, t_2)$. On the other hand, w is not opaque since t_1 reads the old value of v_1 (before t_2 commits) and the new value of v_2 (committed by t_2).

2.2 Transactional memories

We consider thread programs as our basic sequential unit of computations. We express thread programs as infinite binary trees on commands. This 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 easily different retry mechanisms of TMs, e.g., retry the same transaction until it succeeds or try another transaction after an abort. We use a set of thread programs to define a multithreaded program. Formally, a *thread program* θ on a set C of commands is a function $\theta : \mathbb{B}^* \to C$. We write Θ for the set of thread programs. A *(multithreaded) program* p on n threads and k variables is an n-tuple $p = \langle \theta^1, \dots, \theta^n \rangle$ of thread programs on C. Figure 1(a) shows an example program on two threads and two variables. Let $P^{n,k}$ be the set of all programs on n threads and k variables. Let P be the set of all programs.

We define a transactional memory as an abstract function that takes as input a program, and produces a set of infinite words. Formally, a *transactional memory (TM)* is a function $M : P \to 2^{\hat{S}^\omega}$. A transactional memory M *ensures strict serializability (resp. opacity) for all programs with* n *threads and* k *variables* if for every program $p \in P^{n,k}$, every word $w \in M(p)$ is strictly serializable (resp. opaque). Moreover, a transactional memory M *ensures strict serializability* (resp. *opacity*) if it ensures strict serializability (resp. opacity) for all programs with an arbitrary number of threads and variables.



(a) An example program on two threads and two variables

(b) Interaction in the model

Figure 1. Our framework of transactional memory

3. Transactional memory algorithms

We use state transition systems to define TM. A TM algorithm is a family of TM transition systems, one for n threads and k variables, for every n and k. A TM transition system consists of a set of states, an initial state, an extended set of commands depending on the underlying TM, a pending function, and a transition relation between the states. The extended commands include the set C of commands, and TM specific additional commands. For example, a given TM may require that a thread locks a variable before writing to the variable, or that a thread validates the variables read in a transaction, before accessing a new variable. Every extended command is assumed to execute atomically. The pending function represents the pending command of a thread in a state, and ensures that if a thread has not finished the execution of a particular command, then no other command is executed by the thread.

A TM algorithm interacts with a program and a scheduler (see Fig. 1(b)). The scheduler chooses a thread, which determines the next command to be executed. The TM transition system decides whether the command can be executed in a single atomic step, or in several atomic steps (using additional extended commands), or has to be aborted. The TM algorithm gives back to the program a response. The response is $\bot$ if the TM algorithm needs additional steps to complete the command, 0 if the TM algorithm needs to abort the transaction, and 1 if the TM algorithm has completed the command. Given a program, a scheduler, and a TM transition system, we get a run. Projecting the run to the set of successful statements (that is, aborts, and statements that get response 1) gives a word in $\hat{S}^\omega$. We describe the language of a TM transition system as the set of words on $\hat{S}^\omega$ that it can produce for any program and any scheduler.

Formally, a *scheduler* σ on T is a function $\sigma : \mathbb{N} \to T$. We define a *TM algorithm* A as a family of *TM transition systems* $A^{n,k} = \langle Q, q_{init}, D, \pi, \delta \rangle$ for each n and k, where Q is a set of states, q_{init} is the initial state, D is the set of extended commands with $C \subseteq D$, the function $\pi : Q \times T \to C \cup \{\bot\}$ represents the pending command in a state for a thread , and $\delta \subseteq Q \times \hat{C} \times \hat{S}_D \times Resp \times Q$ is the deterministic or non-deterministic transition relation, where $\hat{S}_D = (D \cup \{\mathsf{abort}\}) \times T$ and $Resp = \{\bot, 0, 1\}$. The transition relation δ and the pending function π obey the following rules:

1. For all threads $t \in T$, we have $\pi(q_{init}, t) = \bot$.

2. For all states $q, q' \in Q$ such that there exists an incoming transition $(q, c, (d, t), r, q') \in \delta$ to q', if $r = \bot$, then $\pi(q', t) = c$, otherwise $\pi(q', t) = \bot$.

3. For all states $q, q' \in Q$ such that there exists an incoming transition $(q, c, (d, t), r, q') \in \delta$ to q', then $\pi(q', u) = \pi(q, u)$ for all threads $u \neq t$.

4. For all states q and all threads t, if $\pi(q,t) = c$ where $c \neq \bot$, then for all outgoing transitions $(q, c_1, (d,t), r, q') \in \delta$ from q, we have $c_1 = c$.
5. For all states q and all threads t, if $\pi(q,t) = \bot$, then there exists an outgoing transition $(q, c, (d,t), r, q' \in \delta$ from q for every command $c \in C$.
6. For all $q \in Q$, for all transitions $(q, c, (d,t), r, q') \in \delta$, we have $d = \mathsf{abort}$ if and only if $r = 0$.

Note that the rules above restrict the transition relation and the pending function π such that π is unique. A command c is *enabled* in a state q for thread t if $\pi(q,t) \in \{\bot, c\}$ (i.e., either no command is pending, or c itself is pending). In a deterministic transition relation δ, a command c is *abort enabled* in a state q for thread t if c is enabled in q for thread t and there is no transition $(q, c, (d,t), r, q') \in \delta$ such that $d \in D$. A transition relation δ is *deterministic* if for all $q \in Q$ and $(c,t) \in S$, if $(q, c, (d_1,t), r_1, q_1) \in \delta$ and $(q, c, (d_2,t), r_2, q_2) \in \delta$, then $d_1 = d_2$, $r_1 = r_2$, and $q_1 = q_2$. Unless otherwise stated, TM transition systems have deterministic transition relations. We shall use nondeterministic TM transition systems later to describe reference TM algorithms.

Let $p = \langle \theta^1, \ldots, \theta^n \rangle$ be a program in $P^{n,k}$. Let σ be a scheduler on n threads. A *run* $\rho = \langle q_0, l_0, (d_0, t_0), r_0 \rangle \langle q_1, l_1, (d_1, t_1), r_1 \rangle \ldots$ of a TM transition system $A^{n,k}$ with scheduler σ on program p is an infinite sequence of tuples of states, program locations, statements, and responses, where $l_j = \langle l_j^1, \ldots, l_j^n \rangle \in (\mathbb{B}^*)^n$ for all $j \geq 0$ and the following hold: (i) $q_0 = q_{init}$ and $l_0 = \langle \epsilon, \ldots, \epsilon \rangle$, (ii) for all $j \geq 0$, there exists a transition $(q_j, c_j, (d_j, t_j), r_j, q_{j+1}) \in \delta$ such that $t_j = \sigma(j)$ and $c_j = \theta^{t_j}(l_j^{t_j})$ and for all $t \in T$, we have $l_{j+1}^t = l_j^t$ if either $t \neq t_j$ or $r_j = \bot$, and $l_{j+1}^t = l_j^t \cdot r_j$ otherwise. Given a scheduler and a program, there is exactly one run for a deterministic TM transition system $A^{n,k}$, whereas there is at least one run for a nondeterministic TM transition system. We say that a statement $s_j \in \hat{S}$ is *successful* in the run $\rho = \langle q_0, l_0, s_0, r_0 \rangle \langle q_1, l_1, s_1, r_1 \rangle \ldots$ if (i) $r_j \in \{0, 1\}$, or (ii) $r_k = 1$ with $j < k$ and $r_{j+1} \ldots r_{k-1}$ are all equal to $\bot$. We define the *language* $L(A^{n,k})$ of $A^{n,k}$ as the set of all infinite words $w \in \hat{S}^\omega$ such that w is the sequence of all successful statements in a run of $A^{n,k}$ with some scheduler on n threads, on some program on n threads and k variables. For a TM algorithm A, we require that for $n \leq n'$ and $k \leq k'$, the language $L(A^{n,k}) \subseteq L(A^{n',k'})$.

A TM algorithm A defines a transactional memory M such that for all n and k, for every program p in $P^{n,k}$ and every word $w \in \hat{S}^\omega$, we have $w \in M(p)$ iff there exists a scheduler σ on T and a corresponding run ρ of $A^{n,k}$ with σ on p such that w is the sequence of all successful statements in ρ. It follows that a TM defined by a TM algorithm A ensures strict serializability (resp. opacity) for all programs with n threads and k variables iff all words in $L(A^{n,k})$ are strictly serializable (resp. opaque).

In the following sections, we describe different transactional memories as TM algorithms. To simplify the description, we view a state q of the corresponding TM transition systems as an n-tuple $\langle q^1 \ldots q^n \rangle$, where each component q^t corresponds to a thread t and is called the *thread state* of t.

3.1 The sequential TM

To keep our first example simple, we describe a sequential TM. The sequential TM executes the transactions sequentially (as ideally suited for a uniprocessor). We define the sequential TM M_{seq} using a sequential TM algorithm A_{seq}. The sequential TM transition system $A_{seq}^{n,k}$ for n threads and k variables is given by the tuple $\langle Q, q_{init}, D, \pi, \delta \rangle$. The thread state q^t of thread t is in $\{\mathsf{T}, \mathsf{F}\}$. If a thread t has an unfinished transaction in a state q, then the thread state q^t is T, and F otherwise. The initial state $q_{init} = \langle \mathsf{F}, \ldots, \mathsf{F} \rangle$. The set of extended commands is $D = C$. A transition $(q_1, c, (d,t), r, q_2)$ is in δ if c is enabled in q_1 for thread t and one of the following holds:
1. Read (resp. write). (i) $c = (\mathsf{read}, v)$ (resp. $c = (\mathsf{write}, v)$) and $d = c$ and $r = 1$, and (ii) $q_1^u = \mathsf{F}$ for all $u \neq t$, and (iii) $q_2^t = \mathsf{T}$ and $q_2^u = q_1^u$ for all $u \neq t$. When a thread reads (resp. writes) a variable, if the state of all other threads is F, then the state of t is set to T.
2. Commit. (i) $c = \mathsf{commit}$ and $d = c$ and $r = 1$, and (ii) $q_1^u = \mathsf{F}$ for all $u \neq t$, and (iii) $q_2^t = \mathsf{F}$ and $q_2^u = q_1^u$ for all $u \neq t$. When a thread commits, if the state of all other threads is F, then the state of t is set to F.

A transition $(q_1, c, (\mathsf{abort}, t), 0, q_2)$ is in δ if c is abort enabled in q_1 for thread t and $q_2 = q_1$.

3.2 The two-phase locking TM

Our second example of a TM algorithm is based on two-phase locking (2PL) protocol, commonly used in database transactions. Every transaction locks the variables it reads or writes before accessing them, and releases all acquired locks during the commit. A shared lock is acquired for reading, and an exclusive lock is acquired for writing. We define the 2PL TM M_{2PL} using a 2PL TM algorithm A_{2PL}. The 2PL TM transition system $A_{2PL}^{n,k}$ for n threads and k variables is given by the tuple $\langle Q, q_{init}, D, \pi, \delta \rangle$. The thread state q^t of thread t is a pair $\langle rs, ws \rangle$, where $rs \subseteq V$ is the set of variables locked by t in shared mode, and $ws \subseteq V$ is the set of variables locked in exclusive mode. For every thread, the initial thread state of thread t is $q_{init}^t = \langle \emptyset, \emptyset \rangle$. The set of extended commands is $D = C \cup (\{\mathsf{rlock}, \mathsf{wlock}\} \times V)$. A transition $(q_1, c, (d,t), r, q_2)$ is in δ if c is enabled in q_1 for thread t and one of the following holds:
1. Read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \in ws_1^t \cup rs_1^t$, and (iii) $q_2 = q_1$. When a thread reads a variable that already exists in the read set or the write set of the thread, then the state does not change.
2. Write. (i) $c = (\mathsf{write}, v)$ and $d = c$ and $r = 1$, and (ii) $v \in ws_1^t$, and (iii) $q_2 = q_1$. When a thread writes to a variable that already exists in the write set of the thread, then the state does not change.
3. Read Lock. (i) $c = (\mathsf{read}, v)$ and $d = (\mathsf{rlock}, v)$ and $r = \bot$, and (ii) $v \notin rs_1^t$ and $v \notin ws_1^u$ for all threads u, and (iii) $rs_2^t = rs_1^t \cup \{v\}$, and (iv) $q_2^u = q_1^u$ for all threads $u \neq t$. When a thread t reads v, and v is not in the read set of t, and v is not in the write set of any thread, then v is added to the read set of t.
4. Write Lock. (i) $c = (\mathsf{write}, v)$ and $d = (\mathsf{wlock}, v)$ and $r = \bot$, and (ii) $v \notin ws_1^t$ and $v \notin rs_1^u \cup ws_1^u$ for all threads $u \neq t$, and (iii) $rs_2^t = rs_1^t \cup \{v\}$, and (iv) $q_2^u = q_1^u$ for all threads $u \neq t$. When a thread writes v, and v is not in the write set of t, and v is not in the read set or write set of any thread other than t, then v is added to the write set of t.
4. Commit. (i) $c = \mathsf{commit}$ and $d = c$ and $r = 1$, and (ii) $rs_2^t = \emptyset$, and $ws_2^t = \emptyset$, and (iii) for all threads $u \neq t$, we have $q_2^u = q_1^u$. When a thread commits, the read set and the write set are changed to empty.

A transition $(q_1, c, (\mathsf{abort}, t), 0, q_2)$ is in δ if c is abort enabled in q_1 for thread t and $rs_2^t = \emptyset$ and $ws_2^t = \emptyset$, and $q_2^u = q_1^u$ for all threads $u \neq t$.

3.3 The dynamic software transactional memory

Dynamic software transactional memory (DSTM) [HLMS03] is one of the most popular STM algorithms. The algorithm exists in several flavors. In this work, we focus on one of them, called *invisible read DSTM*, where the transactions require ownership of variables only for writing. The readers are not visible to the writers. Upon reading, a transaction validates its read set in order to ensure opacity. In our work, we ignore optimizations like early release

possible in DSTM. We model the situation of a transaction aborting another transaction by allowing each transaction to set an abort flag for other transactions, and requiring that a transaction aborts whenever the abort flag is set for the thread. We define DSTM TM M_{dstm} using a DSTM TM algorithm A_{dstm}. The DSTM TM transition system $A^{n,k}_{dstm}$ for n threads and k variables is given by $\langle Q, q_{init}, D, \pi, \delta\rangle$. A thread state q^t of thread t is defined as a 3-tuple $\langle status^t, rs^t, os^t\rangle$, where $status^t \in \{\mathsf{aborted}, \mathsf{validated}, \mathsf{invalid}, \mathsf{finished}\}$ is the status of thread t, and $rs^t \subseteq V$ is the read set of thread t, and $os^t \subseteq V$ is the ownership set of thread t. For every thread, the initial thread state of thread t is $q^t_{init} = \langle \mathsf{finished}, \emptyset, \emptyset\rangle$. The set of extended commands is $D = C \cup (\{\mathsf{own}\} \times V) \cup \{\mathsf{validate}\}$. A transition $(q_1, c, (d, t), r, q_2)$ is in δ if c is enabled in q_1 for thread t and one of the following holds:

1. Local read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \in os^t_1$ and $status^t_1 \neq \mathsf{aborted}$, and (iii) $q_2 = q_1$. When a thread reads v such that the read is not global, the state does not change.

2. Global read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \notin os^t_1$ and $status^t_1 = \mathsf{finished}$, and (iii) $rs^t_2 = rs^t_2 \cup \{v\}$ and $os^t_2 = os^t_1$ and $status^t_2 = status^t_1$, and (iv) $q^u_2 = q^u_1$ for all threads $u \neq t$. When a thread reads v globally, if the status of the thread is $\mathsf{finished}$, then v is added to the read set of the thread.

3. Own. (i) $c = (\mathsf{write}, v)$ and $d = (\mathsf{own}, v)$ and $r = \bot$, and (ii) $status^t_1 \neq \mathsf{aborted}$, and (iii) $rs^t_2 = rs^t_1$ and $os^t_2 = os^t_1 \cup \{v\}$ and $status^t_2 = status^t_1$, and (iv) for all threads $u \neq t$, if $v \in os^u_1$, then $status^u_2 = \mathsf{aborted}$, and $os^u_2 = \emptyset$, and $rs^u_2 = \emptyset$, otherwise $status^u_2 = status^u_1$ and $os^u_2 = os^u_1$ and $rs^u_2 = rs^u_1$. When a thread writes to v, if the status of the thread is not $\mathsf{aborted}$, then the variable v is added to the owned set of the thread, and if some thread owns v, then the status of that thread is set to $\mathsf{aborted}$ and its read set and own set are set to empty.

4. Write. (i) $c = (\mathsf{write}, v)$ and $d = c$ and $r = 1$, and (ii) $status^t_1 \neq \mathsf{aborted}$ and $v \in os^t_1$, and (iii) $q^u_2 = q^u_1$ for all $u \in T$. When a thread writes to v, if the status is not aborted and v is in the own set of the thread, then the state does not change.

5. Validate. (i) $c = \mathsf{commit}$ and $d = \mathsf{validate}$ and $r = \bot$, and (ii) $status^t_1 = \mathsf{finished}$ and $status^t_2 = \mathsf{validated}$, and (iii) for all threads $u \neq t$, we have $rs^u_2 = rs^u_1$ and $os^u_2 = os^u_1$, and if $rs^t_1 \cap os^u_1 \neq \emptyset$, then $status^u_2 = \mathsf{aborted}$, else $status^u_2 = status^t_1$. When a thread t commits, if the status is $\mathsf{finished}$, then the status is changed to $\mathsf{validated}$, and for all threads u whose own set intersects with the read set of t, the status of u is changed to $\mathsf{aborted}$.

6. Commit. (i) $c = \mathsf{commit}$ and $d = c$ and $r = 1$, and (ii) $status^t_1 = \mathsf{validated}$, and (iii) $os^t_2 = \emptyset$, and $rs^t_2 = \emptyset$ and $status^t_2 = \mathsf{finished}$, and (iv) for all threads $u \neq t$, we have $rs^u_2 = rs^u_1$ and $os^u_2 = os^u_1$, and if $rs^u_1 \cap os^t_1 \neq \emptyset$, then $status^u_2 = \mathsf{invalid}$, else $status^u_2 = status^u_1$. When a thread t commits, if the status is $\mathsf{validated}$, then the own set and read set of t are set to empty and the status is set to $\mathsf{finished}$, and the status of threads, whose read set intersects with the own set of t, is set to $\mathsf{invalid}$.

A transition $(q_1, c, (\mathsf{abort}, t), 0, q_2)$ is in δ if the command c is abort enabled in q_1 for thread t, and $status^t_2 = \mathsf{finished}$, and $rs^t_2 = \emptyset$ and $os^t_2 = \emptyset$, and $q^u_2 = q^u_1$ for all threads $u \neq t$.

3.4 The TL2 transactional memory

Transactional locking 2 (TL2) [DSS06] is a TM that uses global version numbers to ensure correctness. Version numbers allow efficient read set validation in a distributed setting. We model version numbers using modified sets for each thread. When a transaction commits, it adds its write set to the modified set of every thread with an unfinished transaction. We define the TL2 TM M_{TL2} using the TL2 TM algorithm as A_{TL2}. The TL2 TM transition system $A^{n,k}_{TL2}$ for n threads and k variables is given by the tuple $\langle Q, q_{init}, D, \pi, \delta\rangle$. A thread state q^t of t in the TL2 algorithm is defined as a 5-tuple $\langle status^t, rs^t, ws^t, ls^t, ms^t\rangle$, where $status^t \in \{\mathsf{validated}, \mathsf{finished}\}$, $rs^t \subseteq V$ is the read set, $ws^t \subseteq V$ is the write set, $ls^t \subseteq V$ is the lock set, and $ms^t \subseteq V$ is the modified set. The initial thread state $q^t_{init} = \langle \mathsf{finished}, \emptyset, \emptyset, \emptyset, \emptyset\rangle$ for all threads $t \in T$. The set of extended commands is $D = C \cup (\{\mathsf{lock}\} \times V) \cup \{\mathsf{validate}\}$. A transition $(q_1, c, (d, t), r, q_2)$ is in δ if c is enabled in q_1 for t and one of the following holds:

1. Local read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \in ws^t_1$ and $q_2 = q_1$. When a thread reads v such that the read is not global, the state does not change.

2. Global read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \notin ws^t_1$ and $v \notin ms^t_1$, and (iii) $rs^t_2 = rs^t_1 \cup \{v\}$ and $ls^t_2 = ls^t_1$ and $ws^t_2 = ws^t_1$ and $ms^t_2 = ms^t_1$ and $status^t_2 = status^t_1$, and (iv) for all threads $u \neq t$, we have $q^u_2 = q^u_1$. When a thread reads v and the read is global, if the variable is not in the modified set, then v is added to the read set.

3. Write. (i) $c = (\mathsf{write}, v)$ and $d = c$ and $r = 1$, and (ii) $ws^t_2 = ws^t_1 \cup \{v\}$ and $ls^t_2 = ls^t_1$ and $rs^t_2 = rs^t_1$ and $ms^t_2 = ms^t_1$ and $status^t_2 = status^t_1$, and (iv) $q^u_2 = q^u_1$ for all threads $u \neq t$. When a thread writes to v, the variable v is added to its write set.

4. Lock. (i) $c = \mathsf{commit}$ and $d = (\mathsf{lock}, v)$ and $r = \bot$, and (ii) $status^t_1 = \mathsf{finished}$ and $v \in ws^t_1$ and (iii) there is no thread $u \in T$ such that $v \in ls^t_1$, and (iv) $ls^t_2 = ls^t_1 \cup \{v\}$ and $rs^t_2 = rs^t_1$ and $ws^t_2 = ws^t_1$ and $ms^t_2 = ms^t_1$ and $status^t_2 = status^t_1$, and (v) for all threads $u \neq t$, we have $q^u_2 = q^u_1$. When a thread t commits, if the status is $\mathsf{finished}$ and v is in the write set of t and v is not in the lock set of any thread, then v is added to the lock set of t.

5. Validate. (i) $c = \mathsf{commit}$ and $d = \mathsf{validate}$ and $r = \bot$, and (ii) $status^t_1 = \mathsf{finished}$ and $rs^t_1 \cap ms^t_1 = \emptyset$ and $ws^t_1 = ls^t_1$ and for all threads $u \neq t$, we have $rs^t_1 \cap ls^u_1 = \emptyset$, and (iii) $status^t_2 = \mathsf{validated}$ and $ls^t_2 = ls^t_1$ and $rs^t_2 = rs^t_1$ and $ws^t_2 = ws^t_1$ and $ms^t_2 = ms^t_1$, and (iv) $q^u_2 = q^u_1$ for all threads $u \neq t$ When a thread commits, if the status is $\mathsf{finished}$, and the read set does not intersect with the modified set, and the write set is equal to the lock set, and the read set does not intersect with the lock set of any other thread, then the status is set to $\mathsf{validated}$.

6. Commit. (i) $c = \mathsf{commit}$ and $d = c$ and $r = 1$, and (ii) $status^t_1 = \mathsf{validated}$, and (iii) $rs^t_2 = ls^t_2 = ws^t_2 = ms^t_2 = \emptyset$ and $status^t_2 = \mathsf{finished}$, and (iv) for all threads $u \neq t$, we have $rs^u_2 = rs^u_1$ and $ws^u_2 = ws^u_1$ and $ls^u_2 = ls^u_1$ and $status^u_2 = status^u_1$. (v) for all threads $u \neq t$ such that $rs^u_1 \cup ws^u_1 \neq \emptyset$, we have $ms^u_2 = ms^u_1 \cup ws^t_1$. When a thread t commits, if the status is $\mathsf{validated}$, then the status is changed to $\mathsf{finished}$, and the read, write, lock, and modified sets are set to empty, and all variables written by t are added to the modified sets of all threads that have an unfinished transaction.

A transition $(q_1, c, (\mathsf{abort}, t), 0, q_2)$ is in δ if the command c is abort enabled in q_1 for thread t, and $status^t_2 = \mathsf{finished}$, and $rs^t_2 = ws^t_2 = ls^t_2 = ms^t_2 = \emptyset$, and $q^u_2 = q^u_1$ for all threads $u \neq t$.

3.5 The optimistic concurrency control TM

We now discuss a common concurrency protocol used in databases. It was proposed by Kung et al. [KR81], and is called optimistic concurrency control (OCC). OCC executes the transactions of the threads without any synchronization. Before committing, every transaction chooses a sequence number and validates its read set. Transactions commit in the order of sequence numbers, which we model using precedence sets, similar to the way we modeled version numbers using modified sets in the TL2 TM algorithm.

We define the OCC TM M_{occ} using an OCC TM algorithm A_{occ}. We refer to the OCC TM transition system with n threads and k variables as $A^{n,k}_{occ}$. The formal definition of the transition system can be obtained from the original algorithm, as we did in the previous examples.

Table 1 shows runs with different schedules on the program in Figure 1(a), for each TM algorithm described above.

Table 1. Examples of runs and words in the language of different TM algorithms. Notation: $r = \mathsf{read}$, $w = \mathsf{write}$, $c = \mathsf{commit}$, $a = \mathsf{abort}$, $l = \mathsf{lock}$, $o = \mathsf{own}$, $v = \mathsf{validate}$, $k = \mathsf{chklock}$, $s = \mathsf{serialize}$. Command (c, t) is written as c_t.

TM	Scheduler output	The sequence $s_0 s_1 \dots$ in the run of $L(A)$	The word for the run of $L(A)$
seq	$11122\dots$	$(r,1)_1, (w,2)_1, c_1, (w,1)_2, c_2 \dots$	$(r,1)_1, (w,2)_1, c_1, (w,1)_2, c_2 \dots$
	$112122\dots$	$(r,1)_1, (w,2)_1, a_2, c_1, (w,1)_2, c_2 \dots$	$(r,1)_1, (w,2)_1, a_2, c_1, (w,1)_2, c_2 \dots$
2PL	$111112\dots$	$(l,1)_1, (r,1)_1, (l,2)_1, (w,2)_1, c_1, (l,2)_2 \dots$	$(r,1)_1, (w,2)_1, c_1 \dots$
	$1211112\dots$	$(l,1)_1, a_2, (r,1)_1, (l,2)_1, (w,1)_1, c_1, (l,2)_2 \dots$	$a_2, (r,1)_1, (w,2)_1, c_1 \dots$
dstm	$12211112\dots$	$(r,1)_1, (o,1)_2, (w,1)_2, (o,2)_1, (w,2)_1, v_1, c_1, a_2 \dots$	$(r,1)_1, (w,1)_2, (w,2)_1, c_1, a_2 \dots$
	$12222111\dots$	$(r,1)_1, (o,1)_2, (w,1)_2, , v_2, c_2, (o_2)_1, (w,2)_1, a_1 \dots$	$(r,1)_1, (w,1)_2, c_2, (w,2)_1, a_1 \dots$
TL2	$11211122212\dots$	$(r,1)_1, (w,2)_1, (w,1)_2, (l,2)_1, k_1, v_1, (l,1)_2, k_2, v_2, c_1, c_2 \dots$	$(r,1)_1, (w,2)_1, (w,1)_2, c_1, c_2 \dots$
	$112121222\dots$	$(r,1)_1, (w,2)_1, (w,1)_2, (l,2)_1, (l,1)_2, a_1, k_2, v_2, c_2 \dots$	$(r,1)_1, (w,2)_1, (w,1)_2, a_1, c_2 \dots$
occ	$1211212\dots$	$(r,1)_1, (w,1)_2, (w,2)_1, s_1, s_2, c_1, c_2 \dots$	$(r,1)_1, (w,1)_2, (w,2)_1, c_1, c_2 \dots$
	$1221112\dots$	$(r,1)_1, (w,1)_2, s_2, (w,2)_1, s_1, a_1, c_2 \dots$	$(r,1)_1, (w,1)_2, (w,2)_1, a_1, c_2 \dots$

4. Reduction theorem for safety

We present a reduction theorem for strict serializability and opacity. The theorem states that if a TM ensures strict serializability (resp. opacity) for all programs on two threads and two variables, then the TM ensures strict serializability (resp. opacity). The reduction theorem relies on certain structural properties of transactional memories. These properties are satisfied by all TMs that we discussed in the previous section. For every property, we also give more details on why the mentioned TMs satisfy these properties. Note that the properties are sufficient (and not necessary) conditions for the reduction theorem to hold.

We define four structural properties for TMs. Let M be a transactional memory. Let p be a program on n threads and k variables. Let w be a finite prefix of a word in $M(p)$.

P1. *Transaction projection.* Aborting and unfinished transactions can influence other transactions only by forcing them to abort. Thus, removing all aborting transactions and some of the unfinished transactions do not change the response of the TM to the remaining statements. Formally, let X be the set of transactions in w. We define the *transaction projection* of w on $X' \subseteq X$ as the subsequence of w that contains every statement of all transactions in X'. The property P1 states that the transaction projection of w on X', where X' contains all committing transactions, no aborting transactions, and any subset of the unfinished transactions in w, is in $M(p')$ for some program p'. For instance, a TM satisfies P1 if for every thread t: (i) whenever a statement of an aborting or unfinished transaction of thread t changes the state of another thread u, then u cannot commit, and (ii) upon an abort, the state of t is reset to the initial thread state of t.

P2. *Thread symmetry.* For non-overlapping transactions, the TM is oblivious to the identity of the thread executing the transaction. The property P2 states that if (i) w have 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 word w, either $x <_w y$ or $y <_w x$, then the word w' obtained by renaming all transactions of thread u to be from thread t is a finite prefix of a word in $M(p')$ for some program p' on $n-1$ threads and k variables. For instance, a TM satisfies P2 if (i) the thread state is set to the initial thread state upon a commit, and (ii) the transition relation is identical for all threads.

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 w on $V' \subseteq V$ as the subsequence of w that contains all commit and abort statements, and all read and write statements to variables in V'. The property P3 states that if w has no aborting transactions, then for all $V' \subseteq V$, the variable projection of w on V' is in $M(p')$, where p' is obtained by removing all read and write statements to variables in $V \setminus V'$ from all thread programs in p. For instance, a TM satisfies P3 if reading or writing a variable does not remove a conflict on other variables. All TMs we know of satisfy P3 as they track every variable accessed by every thread independently.

P4. *Monotonicity.* If a word is allowed by the TM, then more sequential forms of the word are also allowed. Formally, let $F \subseteq S^*$ be the set of opaque (resp. strict serializable) words with exactly one unfinished transaction. We define a function $seq : F \to 2^F$ such that if $w_2 \in seq(w_1)$ and y is the unfinished transaction in w_1, then (i) $com(w_2)$ is sequential and strictly equivalent to $com(w_1')$, and (ii) all statements of y in w_1' occur in w_2 in some order such that order of all conflicts of global reads in y with other transactions in w_1' is preserved, where w_1' is obtained from w_1 by adding for every transaction x that commits before y in w, a write of an auxiliary variable v_{xy} to x, and a read of v_{xy} to y. (These variables are introduced to maintain the order of transactions.) The monotonicity property for opacity (resp. strict serializability) states that if $w = w' \cdot s$, where $w' \in F$, and s is not an abort, and s is a statement of the unfinished transaction in w', then for every word $w_2 \in seq(w')$, the word $w_2 \cdot s$ is a finite prefix of a word in $M(p')$ for some program p'. For instance, a TM satisfies P4 if it is unfinished commutative and commit commutative. A TM is *unfinished commutative* if for all words $w_p, w_q, w_s \in S^*$, if the word $w_p \cdot w_q \cdot s \cdot w_s$ is a finite prefix of a word in $M(p)$, where s is a global read and no statement in w_q conflicts with s, then $w_p \cdot s \cdot w_q \cdot w_s$ is a finite prefix of a word in $M(p')$ for some program p'. A TM is *commit commutative* if for all words $w_p, w_q, w_s \in S^*$, if $w_p \cdot w_q \cdot s \cdot w_s$ is a finite prefix of a word in $M(p)$, where s is a commit of some transaction x and no statement in w_q conflicts with s, then the word $w_p \cdot x \cdot w_q' \cdot w_s$ is a finite prefix of a word in $M(p')$ for some program p', where w_q' is the word obtained by removing transaction x from w_q. The idea is that with these commutativity rules, an interleaved word can be made sequential. The TMs, 2PL, DSTM, TL2 and OCC are unfinished commutative and commit commutative, and thus satisfy monotonicity.

Theorem 1. If a TM M ensures strict serializability (resp. opacity) for all programs on two threads and two variables, and satisfies the properties P1, P2, P3, and P4 for opacity (resp. strict serializability), then M ensures strict serializability (resp. opacity).

Proof. We prove the theorem for strict serializability. A similar proof holds for opacity. The proof is by contradiction. Let p be a program in $P^{n,k}$. Let w be a word in $M(p)$ such that w is not strictly serializable. Let w_p be the longest finite prefix of w such that w_p is strictly serializable and let $w_1 = w_p \cdot s$, where

$s = (c, t)$ is a statement of transaction x. Let X be the set of committed transactions in w_p. By property P1, there exists a word w_2 generated by projecting w_1 to $X \cup \{x\}$ such that w_2 is a finite prefix of a word in $M(p_2)$ for some program p_2. We note that $w_2 = w'_p \cdot s$ and w'_p is strictly serializable and w_2 is not strictly serializable. So, using property P4 for strict serializability, there exists a word $w''_p \in seq(w'_p)$ such that the word $w_3 = w''_p \cdot s$ is a finite prefix of a word in $M(p_2)$. In w_3 only one transaction, x, does not execute sequentially. Using property P2, we rename the threads for the transactions in w_3. We let all transactions except x to be executed by thread u. Let this renaming give word w_4. We note that the last statement of x is a commit. As w_4 is not strictly serializable, we know (by the definition of conflict) that one of the following holds: (i) $s_1 = ((\mathsf{read}, v_1), t)$ and $s_2 = ((\mathsf{read}, v_2), t)$ are global reads of transaction x such that some transaction y of thread u writes to v_1 and some transaction y' of u with $y' = y$ or $y <_{w_4} y'$ writes to v_2 and both commit between s_1 and s_2, (note that y and y' cannot overlap due to the structure of w_4,) or (ii) $s_1 = ((\mathsf{read}, v_1), t)$ is a global read of transaction x such that some transaction y of thread u writes to v_1 and commits after s_1, and there is a committing transaction y' with $y' = y$ or $y <_{w_4} y'$ which has a command (read, v_2) or (write, v_2), and x also writes to v_2. (Note that v_1 may be same as v_2). Let w_5 be a variable projection of w_4 on $\{v_1, v_2\}$. We know that w_5 is a finite prefix of a word in $M(p_5)$ for some program p_5 on two threads and two variables, by property P3. Also, we note that w_5 is not strictly serializable. As M ensures strict serializability for all programs on two threads and two variables, we get a contradiction. Thus, there is no such program p_5. This leads us to a contradiction. □

5. The reference TM algorithms

To verify the safety properties of a transactional memory, we take the following approach. We construct a reference TM algorithm for strict serializability (RSS TM algorithm), whose language is exactly the set of all strictly serializable words. Similarly, we construct a reference TM algorithm for opacity (RO TM algorithm), whose language is exactly the set of all opaque words. Then, we show that a given TM defined by a TM algorithm A ensures strict serializability (resp. opacity) iff for all n and k, all words in $L(A^{n,k})$ are in the language of the RSS (resp. RO) TM transition system for n threads and k variables. If the given TM satisfies the structural properties presented in the previous section, it is sufficient to check that all words in $L(A^{2,2})$ are in the language of the RSS (resp. RO) TM transition system for 2 threads and 2 variables.

The key insight that makes our technique work is that the reference TM algorithms for strict serializability and opacity for two threads and two variables can be defined as *finite-state* transition systems. This is not obvious, as threads may be delayed arbitrarily, transactions may contain arbitrarily many statements and may be aborted arbitrarily often. We present the RSS TM transition system first, because it provides the basis for defining the RO TM transition system. Suitable finite-state reference TM transition systems can also be defined for stronger notions of safety, such as the notions described by Scott [Sco06], by modifying the semantics of conflict.

5.1 The reference TM algorithm for strict serializability

The classical approach to checking whether a word is strictly serializable is to construct a directed graph $G = (V, E)$, called the conflict graph [Pap79], of the committing transactions in the word. The conflict graph captures the precedence of the committing transactions based on the conflicts. Given a word $w = s_0 s_1 \ldots$, the transactions in w form the set V of vertices in the conflict graph. There exists an edge from a vertex v_1 to a vertex v_2 if v_2 commits or aborts before v_1 starts, or a statement s_i of v_1 conflicts with a statement s_j of v_2 and $i > j$. The conflict graph G is acyclic iff the word w is strictly serializable. We note that the size of this construction is unbounded. The following parameterized word illustrates the point: $w_m =$((read, v_1), t_1), (((write, v_1), t_2), (commit, t_2))^m, (commit, t_1). The number of vertices in the conflict graph of w_m is $m + 1$. Thus, we cannot aim to create a finite transition system for the RSS TM algorithm using conflict graphs. We give a first finite state representation for the language of strictly serializable words, when transactions may abort. The idea of maximal serializability was also addressed earlier [FR85] for a bounded number of non-aborting transactions with a bounded number of statements per transaction. The idea was built upon a notion of transitive conflicts, which does not hold when transactions may abort.

The key idea to get around the problem of infinite states is to maintain sets called *prohibited read and write sets* for every thread. These sets allow to handle unbounded delay between transactions, as committing transactions store the required information in the sets of other threads. Once a transaction commits or aborts, we need not remember it (unlike conflict graphs). Thus, we need to store information of at most one transaction per thread. The RSS TM transition system is based on the following observation: *Every committing transaction serializes at some point during its execution*. The RSS TM transition system makes a non-deterministic guess of when a transaction serializes. Depending upon the guess, the transition system checks upon the commit of a transaction, whether the commit can be executed, or it needs to abort.

Formally, we define the *RSS TM algorithm* A_{ss} as a family of RSS TM transition systems. The *RSS TM transition system* $A_{ss}^{n,k}$ for n threads and k variables is given by the tuple $\langle Q, q_{init}, D, \pi, \delta \rangle$. The thread state q^t is a 6-tuple of the form $\langle Status^t, rs^t, ws^t, prs^t, pws^t, Preds^t \rangle$, where $Status^t \in \{\mathsf{started}, \mathsf{invalid}, \mathsf{serialized}, \mathsf{finished}\}$ is the status function, $rs^t \subseteq V$ is the read set, $ws^t \subseteq V$ is the write set, $prs^t \subseteq V$ is the prohibited read set, $pws^t \subseteq V$ is the prohibited write set, and $Preds^t \subseteq T$ is the predecessor set for thread t. If $v \in prs^t$ (resp. $v \in pws^t$), then the status of the thread t is set to invalid if t globally reads (resp. writes to) v. If $u \in Preds^t$, then the unfinished transaction of u has to commit before the unfinished transaction of t. The initial thread state q^t_{init} is $\langle \mathsf{finished}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$. The set of extended commands is $D = C \cup \{\mathsf{serialize}\}$. The transition relation δ is nondeterministic. A transition $(q_1, c, (d, t), r, q_2)$ is in δ if c is enabled in q_1 for thread t and one of the following holds.

1. Local read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \in ws^t_1$, and (iii) $q_2 = q_1$. When a thread reads v, if the read is not global, then the state remains unchanged.

2. Global read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \notin ws^t_1$, and (iii) if $status^t_1 = \mathsf{finished}$, then $status^t_2 = \mathsf{started}$, else if $status^t_1 = \mathsf{serialized}$ and $v \in prs^t_1$, then $status^t_2 = \mathsf{invalid}$, else $status^t_2 = status^t_1$, and (iv) $rs^t_2 = rs^t_1 \cup \{v\}$ and $ws^t_2 = ws^t_1$ and $prs^t_2 = prs^t_1$ and $pws^t_2 = pws^t_1$ and $Preds^t_2 = Preds^t_1$, and (v) for all threads $u \neq t$, we have $q^u_2 = q^u_1$. When a thread t reads v globally, v is added to the read set, and if the status of t is finished, then the status of t is changed to started, else if the status of t is serialized and v is in the prohibited read set, then the status of t is changed to invalid.

3. Write. (i) $c = (\mathsf{write}, v)$ and $d = c$ and $r = 1$, and (ii) if $status^t_1 = \mathsf{finished}$, then $status^t_2 = \mathsf{started}$, else if $status^t_1 = \mathsf{serialized}$ and $v \in pws^t_1$, then $status^t_2 = \mathsf{invalid}$, else $status^t_2 = status^t_1$, and (iii) $ws^t_2 = ws^t_1 \cup \{v\}$ and $rs^t_2 = rs^t_1$ and $prs^t_2 = prs^t_1$ and $pws^t_2 = pws^t_1$ and $Preds^t_2 = Preds^t_1$, and (iv) for all threads $u \neq t$, we have $q^u_2 = q^u_1$ When a thread t writes to v, the variable v is added to the write set, and if the status of t is finished, then the status of t is changed to started, else if the status of t is serialized and v is in the prohibited write set, then the status of t is changed to invalid.

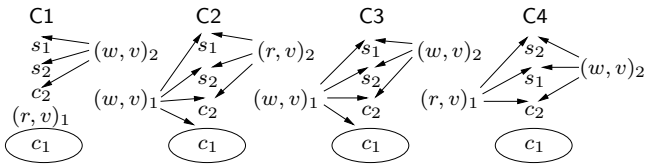


Figure 2. We use the same notation as in Table 1. The commits inside ovals are disallowed by the RSS algorithm. Each condition shows various cases. The arrows represent different possible positions for a command to occur in a given condition.

4. Serialize. (i) $d = \mathsf{serialize}$ and $r = \bot$, and (ii) $status_1^t = \mathsf{started}$, and (iii) $status_2^t = \mathsf{serialized}$ and $rs_2^t = rs_1^t$ and $ws_2^t = ws_1^t$ and $prs_2^t = prs_1^t$ and $pws_2^t = pws_1^t$ and $Preds_2^t = \{u \in T \mid Status_1^u = \mathsf{serialized}\}$, and (iv) for all threads $u \neq t$, we have $q_2^u = q_1^u$. Upon any command of thread t, if the current status of t is started and if the thread t chooses to serialize, then the status of t is set to serialized, and every thread whose status is serialized is added to the predecessor set of t.

5. Commit. (i) $c = \mathsf{commit}$ and $d = c$ and $r = 1$, and (ii) $status_1^t \in \{\mathsf{serialized}, \mathsf{finished}\}$, and (iii) $status_2^t = \mathsf{finished}$ and $rs_2^t = ws_2^t = prs_2^t = pws_2^t = Preds_2^t = \emptyset$, and (iv) for all threads $u \neq t$, we have $rs_2^u = rs_1^u$ and $ws_2^u = ws_1^u$ and $Preds_2^u = Preds_1^u \setminus \{t\}$, and (v) for all threads $u \neq t$, if $u \in Preds_1^t$, then $prs_2^u = prs_1^u \cup ws_1^t$ and $pws_2^u = pws_1^u \cup rs_1^t \cup ws_1^t$, else $prs_2^u = prs_1^u$ and $pws_2^u = pws_1^u$, and (vi) for all threads $u \in Preds_1^t$, set $status_2^u = \mathsf{invalid}$ if $ws_1^u \cap (ws_1^t \cup rs_1^t) \neq \emptyset$, and $status_2^u = status_1^u$ otherwise, and (vii) for all threads $u \notin Preds_1^t$ and $u \neq t$, set $status_2^u = \mathsf{invalid}$ if $ws_1^t \cap rs_1^u \neq \emptyset$, and $status_2^u = status_1^u$ otherwise. When a thread t commits, if the current status of t is serialized or finished, then the following happen: The status of t is set to finished. For every predecessor thread u of t, all variables in the write set of t are added to the prohibited read set and the prohibited write set of u, and all variables in the read set of t are added to the prohibited write set of u. For all predecessor threads u of t such that the write set of u intersects with the read set or write set of t, the status of u is set to invalid. For all threads u that are not predecessors of t such that the read set of u intersects with the write set of t, the status of u is set to invalid.

For every state $q_1 \in Q$, a transition $(q_1, c, (\mathsf{abort}, t), 0, q_2)$ is in δ if $c \in C$ is enabled in q_1 for thread t, and $rs_2^t = ws_2^t = prs_2^t = pws_2^t = Preds_2^t = \emptyset$, and $status_2^t = \mathsf{finished}$, and $q_2^u = q_1^u$ for all threads $u \neq t$.

Note that the non-determinism in the transition relation comes from the serialize command, and the fact that abort is allowed in every state where a command is enabled.

Theorem 2. Given a word w on n threads and k variables, the word w is strictly serializable if and only if $w \in L(A_{ss}^{n,k})$.

Proof. Consider a run ρ of $A_{ss}^{n,k}$. Let w_1 be an arbitrary finite prefix of the sequence of all successful statements in ρ, and let X be the set of finished transactions in w_1. Let w' be the sequential word such that w' is transaction equivalent to w and $x <_{w'} y$ if the serialize command of transaction x comes before that of transaction y in ρ (Note that every non-empty transaction has the serialize command exactly once.) Then, $com(w')$ is strictly equivalent to $com(w_1)$ if for every transaction $x \in X$, either the transaction x does not commit in w_1, or one of the following conditions holds for x (graphically shown in Figure 2):

C1. There exists a transaction y such that x serializes before y and y writes to a variable v and commits, and then x globally reads v.

C2. There exists a transaction y such that x serializes before y and x writes to v and y reads v before x commits, and y commits.

C3. There exists a transaction y such that x serializes before y and both x and y write to a variable v, and y commits before x does.

C4. There exists a transaction y such that x serializes after y and y writes to v and x reads v before y commits, and then y commits.

The RSS TM transition system $A_{ss}^{n,k}$ guarantees by construction, that a transaction x does not commit iff one of the conditions, C1–C4 holds. Hence, every word in $L(A_{ss}^{n,k})$ is strictly serializable.

Conversely, let w be strictly serializable. Consider an arbitrary finite prefix w_2 of w. As w_2 is strictly serializable, there is a sequential word w' such that $com(w')$ is strictly equivalent to $com(w_2)$. Let the committing transactions in the sequential word w' be given by the sequence $x_1 x_2 \ldots$ of transactions. Consider a run ρ of the RSS TM transition system $A_{ss}^{n,k}$ such that w_2 is a finite prefix of all successful statements of ρ, and for all i and j such that $i < j$, the transaction x_i serializes before x_j in ρ. The run ρ exists because (i) the RSS TM transition system guesses every possible serialization for every transaction during its execution, and (ii) given that w_2 is strictly serializable, there is no transaction x in the sequence $x_1 x_2 \ldots$ that satisfies any of the conditions C1–C4, and commits in w_2. Thus, the word w is in the language $L(A_{ss}^{n,k})$. □

5.2 The reference TM algorithm for opacity

Apart from the requirements of the above mentioned reference TM algorithm for strict serializability, opacity requires that even global reads of aborting transactions observe consistent values. It turns out that we can obtain a finite-state representation of the RO TM transition system by slightly modifying our RSS TM transition system.

The RO TM transition system is based on the following observation: *Every committing and aborting transaction should serialize at some point during its execution.* Like the RSS TM transition system, the RO TM transition system makes a non-deterministic guess of when a transaction serializes. In this case, the transition system checks upon every global read and every commit of a transaction, whether the command can be executed or the transaction needs to be aborted. The formalism for *RO TM algorithm* A_{op} and the *RO TM transition system* $A_{op}^{n,k}$ is identical to that of the RSS TM algorithm and the RSS TM transition system. The only difference comes in the transition relation δ, on a global read, and on a serialize command. We obtain the transition relation for $A_{op}^{n,k}$ by replacing rules 2 and 4 of that of $A_{ss}^{n,k}$ by the rules 2a and 4a below.

2a. Global read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \notin ws_1^t$ and $v \notin prs_1^t$, and (iii) $rs_2^t = rs_1^t \cup \{v\}$ and $ws_2^t = ws_1^t$ and $prs_2^t = prs_1^t$ and $pws_2^t = pws_1^t$ and $Preds_2^t = Preds_1^t$, and (iv) if $status_1^t = \mathsf{finished}$, then $status_2^t = \mathsf{started}$, else $status_2^t = status_1^t$, and (v) for all threads $u \neq t$, if $status_1^u = \mathsf{serialized}$ and $t \notin Preds_1^u$ and $v \in ws_1^u$, then $status_2^u = \mathsf{invalid}$, else $status_2^u = status_1^u$, and (vi) for all threads $u \neq t$, if $status_1^u = \mathsf{serialized}$ and $t \notin Preds_1^u$, then $pws_2^u = pws_1^u \cup \{v\}$, else $pws_2^u = pws_1^u$, and (vii) $prs_2^u = prs_1^u$ and $rs_2^u = rs_1^u$ and $ws_2^u = ws_1^u$ and $Preds_2^u = Preds_1^u$ for all threads $u \neq t$. When a thread t reads v globally, if v is not in the prohibited read set, then the following happen: v is added to the read set. If the status of t is finished, then it is changed to started. For every other thread u with status serialized such that t is not a predecessor of u, the variable v is added to the prohibited write set of u, and if v is in the write set of u, then the status of u is set to invalid.

4a. Serialize. (i) $d = \mathsf{serialize}$ and $r = \bot$, and (ii) $status_1^t = \mathsf{started}$, and (iii) if there exists a thread $u \neq t$ such that $status_1^u = \mathsf{started}$ and $rs_1^u \cap ws_1^t \neq \emptyset$, then $status_2^t = \mathsf{invalid}$, else $status_2^t = \mathsf{serialized}$, and (iv) $pws_2^t = pws_1^t \cup V'$ where $V' = \{v \in V \mid v \in rs_1^u$ for some thread $u \neq t$ with $status_1^u = \mathsf{started}\}$, and (v) $rs_2^t = rs_1^t$ and $ws_2^t = ws_1^t$ and $prs_2^t = prs_1^t$ and $Preds_2^t = \{u \in T \mid status_1^u = \mathsf{serialized}\}$, and (vi) for all threads $u \neq t$, if $status_1^u = \mathsf{serialized}$ and $ws_1^u \cap rs_1^t \neq \emptyset$,

Table 2. Time for simulation checking for TM algorithms on a quad dual core 2.8 GHz server with 16 GB RAM. In case simulation holds, we write YES followed by the time required for the simulation. Otherwise, we write NO followed by the counterexample produced, followed by the time required to prove that no simulation exists, followed by the time required to find the counterexample. A '*' for the search for simulation relation means that it does not complete in 2 hours, and we try to find a counterexample.

TM transition system $A^{2,2}$	Number of states	$A^{2,2} \prec A^{2,2}_{ss}$	$A^{2,2} \prec A^{2,2}_{op}$
seq	3	YES, $0.8s$	YES, $0.7s$
$2PL$	99	YES, $13s$	YES, $8s$
$dstm$	944	YES, $127s$	YES, $82s$
$TL2$	11840	YES, $1647s$	YES, $1438s$
occ	4480	YES, $765s$	NO, w_1, $567s$,$4s$
$TL2$ modified	17520	NO, w_2, *, $9s$	NO, w_2, *, $8s$
ss	12346	—	—
op	9202	—	—
Counterexample			
w_1	$(w,1)_2,(r,1)_1,c_2,(r,1)_1$		
w_2	$(w,2)_1,(w,1)_2,(r,2)_2,(r,1)_1,c_2,c_1$		

then $status^u_2 = \mathsf{invalid}$, else $status^u_2 = status^u_1$, and (vii) for all threads $u \neq t$, if $status^u_1 = \mathsf{serialized}$, then $pws^u_2 = pws^u_1 \cup rs^t_1$, else $pws^u_2 = pws^u_1$, and (viii) $prs^u_2 = prs^u_1$ and $rs^u_2 = rs^u_1$ and $ws^u_2 = ws^u_1$ and $Preds^u_2 = Preds^u_1$ for all threads $u \neq t$. Upon any command of thread t, if the current status of t is started, and if the thread chooses to serialize, then the following happen: If there is a thread u with status started such that the read set of u intersects with the write set of t, then the status of t is set to invalid, else the status of t is set to serialized. All variables in read sets of threads with status started are added to the prohibited write set of t. All threads with status serialized are added to the predecessor set of t. For every other thread u, if the status of u is serialized and the write set of u intersects with the read set of t, then the status of u is set to invalid. For every thread u with status serialized, the read set of t is added to the prohibited write set of u.

Theorem 3. Given a word w on n threads and k variables, the word w is opaque if and only if $w \in L(A^{n,k}_{op})$.

5.3 Implementation and simulation checking

A TM defined by a TM algorithm A ensures strict serializability (resp. opacity) iff $L(A^{2,2}) \subseteq L(A^{2,2}_{ss})$ (resp. $L(A^{2,2}) \subseteq L(A^{2,2}_{op})$). As checking language inclusion is PSPACE-hard, we use the common technique of checking for the existence of a simulation relation between both transition systems. The existence of a simulation relation is a sufficient condition for language inclusion. We write $A^{2,2}_1 \prec A^{2,2}_2$ to denote that there exists a simulation relation between $A^{2,2}_1$ and $A^{2,2}_2$. For a TM M defined by a TM algorithm A which satisfies the structural properties of the reduction theorem (Theorem 1), M ensures strict serializability (resp. opacity) if $A^{2,2} \prec A^{2,2}_{ss}$ (resp. $A^{2,2} \prec A^{2,2}_{op}$).

We built an automatic verification tool in C for checking the existence of simulation relations using the quadratic algorithm by Henzinger et al. [HHK95]. The tool is conceived as a platform for the automatic verification of TMs that satisfy the structural properties. We mention that simulation checking requires extra technical care in this scenario due to different extended alphabets in different TMs. The tool takes as input two TM algorithms A_1 and A_2, and checks whether $A^{2,2}_1 \prec A^{2,2}_2$. If the tool fails to find a simulation relation, it attempts to return a finite counterexample $w \in \hat{S}^*$ such that w is a prefix of some word in $L(A^{2,2}_1)$, and w is not a prefix of any word in $L(A^{2,2}_2)$. In certain cases, it is possible that although language inclusion holds, the tool cannot find a simulation relation. Thus, our decision procedure is sound but not complete. For all TM transition systems we considered, our tool terminates after finding a simulation relation, or a counterexample.

The results of our simulation checks are presented in Table 2. Our results demonstrate that all TMs discussed in Section 3 — sequential, 2PL, DSTM, and TL2 TM— are simulated by both reference TM transition systems. As for the OCC TM, it is simulated by the RSS TM transition system, but not by the RO TM transition system. The tool gives a counterexample in the latter case. Our results establish the following theorem.

Theorem 4. The sequential TM, two-phase locking, DSTM, and TL2 ensure opacity. The optimistic concurrency control ensures strict serializability, but not opacity.

Our tool discovered a subtle point in TL2. In the description of the published TL2 algorithm, we found the order of two operations, validating the read set (rvalidate), and checking whether a variable in the read set is locked (chklock), ambiguous. We modeled these operations as two separate atomic operations, such that that chklock happens after rvalidate, to obtain a modified TL2 TM algorithm. The tool found that the modified TL2 TM algorithm is not simulated by the RSS TM transition system, and the tool provided a counterexample. Thus, we conclude that the modified TL2 TM does not ensure strictly serializability, and thus does not ensure opacity. In the published TL2 algorithm, the authors maintain the version number and the lock bit of every variable in the same memory word. This ensures that the two operations chklock and rvalidate execute atomically, and thus they can be executed in any order. So, our experiments discover that the correctness of TL2 is based on the subtle fact that either the version number and the lock bit have to be accessed atomically, or rvalidate has to occur after chklock.

6. Verifying liveness

We define two different notions of liveness, obstruction freedom and livelock freedom, as discussed in the transactional memory literature. A third notion, wait freedom [Her91], implies livelock freedom. Since we will show that none of our example TMs satisfy livelock freedom, they do not satisfy wait freedom either.

A word $w \in \hat{S}^\omega$ is *obstruction free* [HLM03] if for all threads t, if the word w has an infinite number of aborts of t, then w has an infinite number of commits of t or there are infinitely many statements of some thread $u \neq t$. Formally, w is *obstruction free* if $\bigwedge_{t \in T}(\Box\Diamond(\mathsf{abort}, t) \rightarrow \Box\Diamond((\mathsf{commit}, t) \vee \bigvee_{c \in \hat{C}, u \in T \setminus \{t\}}(c, u)))$. This is a Streett condition.

A word $w \in \hat{S}^\omega$ is *livelock free* [AKH03] if the word has an infinite number of commits, or there is a thread t such that t has infinitely many statements and finitely many aborts in w. Formally, w is *livelock free* if $\Box\Diamond(\bigvee_{t \in T}(\mathsf{commit}, t)) \vee \bigvee_{t \in T}(\Box\Diamond(\bigvee_{c \in C}(c, t)) \wedge \Diamond\Box\neg(\mathsf{abort}, t))$. Note that livelock freedom implies obstruction freedom.

A TM M *ensures obstruction freedom (resp. livelock freedom) for all programs with n threads and k variables* if for every program $p \in P^{n,k}$, every word $w \in M(p)$ is obstruction free (resp. livelock free). M *ensures obstruction freedom (resp. livelock freedom)* if M ensures obstruction freedom (resp. livelock freedom) for all programs with an arbitrary number of threads and variables. We use the formalism of TM algorithms to verify liveness properties of TMs. We define a *loop* l in a TM transition sys-

tem $A^{n,k}$ as a finite word $s_0 \dots s_m$ such that there exists a run $\langle q_0, l_0, s_0, r_0\rangle \dots \langle q_m, l_m, s_m, r_m\rangle$ of $A^{n,k}$ such that $q_0 = q_m$.

Note that every word w that is not obstruction free violates at least one of the conjuncts of the Streett condition stated above. Each conjunct (Streett pair) corresponds to one thread. A word w can violate the condition for thread t, only if w has from some point on only statements of t. Note that in this case w trivially satisfies the Streett pairs for other threads. This fact allows us to use a simple model checking procedure, even though obstruction freedom is formally a Streett condition.

In particular, a TM defined by a TM algorithm A ensures obstruction freedom for all programs with n threads and k variables iff there is no loop l in $A^{n,k}$ such that all statements in l are from the same thread, and l contains no commit, and l contains an abort. Similarly, a TM ensures livelock freedom for all programs with n threads and k variables iff there is no loop l in $A^{n,k}$ such l contains no commit, and every thread that has a statement in l, has an abort in l.

6.1 Reduction theorem for liveness

As we did for safety, we state a reduction theorem that proves that it is sufficient to verify liveness of a TM on programs with two threads and one variable to generalize the result to all programs. For this purpose, we describe two more structural properties of TMs. These properties are again satisfied by all TMs that we have discussed. Let $w = w_1 \cdot w_2$ be an infinite word such that w is in $M(p)$ for some program p, and no unfinished transaction in w_1 has a statement in w_2, and all statements in w_2 are from the same thread, and there is no commit command in w_2. For $i \in \{1, 2\}$, let V_i be the variables accessed in w_i.

P5. *Transaction projection.* A thread t running in isolation (no interleaved step from other threads) shall abort repeatedly only if it conflicts with some unfinished transaction. As the number of threads is finite, and a thread can have at most one unfinished transaction, there are infinitely many aborts of t due to a particular thread. The property P5 states that (i) the word $w_1' \cdot w_2$ is in $M(p')$ for some program p', where w_1' is obtained by taking the transaction projection of w_1 on non-aborting transactions, and (ii) if w_1 has no aborting transactions and w_2 reads or writes only one variable, then there exists a word $w' = w_1'' \cdot w_2 \in M(p)$, where w_1'' is obtained by projecting w_1 to transactions of some thread t that has statements in w_1. For instance, a TM satisfies P5 if the state of a thread is reset to the initial state upon an abort command, and every variable accessed by every thread is tracked independently.

P6. *Variable projection.* A thread t running in isolation shall abort repeatedly only if some commands corresponding to some variables are not allowed. As the number of variables is finite, there are infinitely many aborts of t due to a particular variable. The property P6 states that (i) there exists a word $w_1 \cdot w_2' \in M(p')$ for some program p', where w_2' is the variable projection of w_2 on $\{v\}$ for some variable $v \in V_2$, and (ii) if w_1 has no aborting transactions, then the word $w' = w_1' \cdot w_2$ is in $M(p')$ for some program p', where w_1' is the variable projection of w_1 on V_2. For instance, a TM satisfies P6 if the TM tracks every variable accessed by every thread independently.

Theorem 5. If a TM M satisfies properties P5 and P6, and M ensures obstruction freedom for two threads and one variable, then M ensures obstruction freedom.

Proof. Let $w \in M(p)$ be a word on arbitrary number of threads and variables such that w is not obstruction free. As w is not obstruction free, it can be written in the form $w_1 \cdot w_2$ as required by the properties P5 and P6. We can then use these properties to obtain a word w' on two threads and one variable such that $w' \in M(p')$ for some program p'. □

Table 3. Results of model checking liveness on a dual core 2.66GHz desktop PC with 2 GB RAM. The notation is similar to Table 2. The time denotes the time required to prove a liveness property or find a counterexample. The counterexamples obtained are of the form $a \cdot b^\omega$. We write the looping part b here.

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

6.2 Model checking liveness

We built a verification tool to check obstruction freedom and livelock freedom properties for transaction memories defined by TM algorithms. To check obstruction freedom, our tool tries to find a loop l in the TM transition system such that all statements in l are from the same thread, and l has no commit, and l has an abort. If the tool finds such a loop, the loop is a counterexample to obstruction freedom. If the tool does not find a loop, we know that the TM ensures obstruction freedom. Similarly, to check livelock freedom, our tool tries to find a loop l in the TM transition system such that there is no commit in l, and every thread that has a statement in l, has an abort in l.

In this way, our tool provides a platform for TM designers to check which liveness properties are ensured by their TMs. If the liveness property fails, then the tool provides feedback in the form of a run that represents a counterexample. Our results are shown in Table 3. The next theorem follows.

Theorem 6. DSTM ensures obstruction freedom and does not ensure livelock freedom. Sequential TM, two phase-locking, TL2, and optimistic concurrency control do not ensure obstruction freedom.

7. Related Work

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

Our construction of the reference TM algorithms is related to the work of Fle and Roucairol [FR85]. They investigated the set of concurrent traces that are generated by a finite set of iterating transactions. They proved that the language consisting of all traces that are conflict equivalent to a sequential trace is regular. Their results cannot be applied in the presence of aborting transactions, as they require the transitivity of conflicts, which does not hold when transactions may abort.

There has been much research on the formal verification of relaxed memory models and cache-coherence protocols for modern multi-processors, e.g., [HQR99, Qad03, GYS04, BAM07]. In this work, the semantics of a shared memory is generally given by a *memory consistency model*, which defines the possible outcomes of executing a concurrent program. Since our approach specifically targets STM, we use a deferred update semantics rather than a memory consistency model.

8. Conclusion

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

Currently, our framework does not apply when transactions help each other. For instance, we cannot model Fraser's STM [FH07] where threads help each other in order to ensure livelock freedom. For efficient performance during contention, many STM protocols rely on a contention manager, like the *Polite* or *Karma* contention manager of Scherer and Scott [SS05]. In this work, we do not handle some of these contention managers. We plan to extend our work by modeling different contention managers as nondeterministic transition systems. Also, our liveness properties capture deterministic notions. It will be interesting to account for probabilistic means to deal with contention, such as random exponential backoff.

We also assumed that the commands in the extended alphabet, like read, write, validate, commit, executed atomically. So, STM algorithms have to guarantee this level of atomicity to ensure correctness using our methodology. Currently we are extending our work to reason about correctness if the lower level primitives are not atomic.

References

[AKH03] J. H. Anderson, Y. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, pages 75–110, 2003.

[AMP00] R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Information and Computation*, pages 167–188, 2000.

[BAM07] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21, 2007.

[BCG89] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, pages 13–31, 1989.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In *CAV*, pages 68–80. Springer, 1994.

[COP+07] A. Cohen, J. O'Leary, A. Pnueli, M. R. Tuttle, and L. Zuck. Verifying correctness of transactional memories. In *FMCAD*, pages 37–44, 2007.

[DSS06] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208. Springer, 2006.

[EGLT76] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, pages 624–633, 1976.

[FH07] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 2007.

[FR85] M. Flé and G. Roucairol. Maximal serializability of iterated transactions. *Theoretical Computer Science*, pages 1–16, 1985.

[GK08] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, 2008.

[GYS04] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or Not QB: An efficient execution verification tool for memory orderings. In *CAV*, pages 401–413. Springer, 2004.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, pages 124–149, 1991.

[HHK95] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, pages 453–462, 1995.

[HLM03] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529. IEEE Computer Society, 2003.

[HLMS03] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[HM93] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300. ACM Press, 1993.

[HQR99] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In *CAV*, pages 301–315. Springer, 1999.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, pages 213–226, 1981.

[LR07] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489. William Kaufmann, 1971.

[Pap79] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, pages 631–653, 1979.

[Qad03] S. Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Transactions on Parallel and Distributed Systems*, pages 730–741, 2003.

[Sco06] M. L. Scott. Sequential specification of transactional memory semantics. In *TRANSACT*, 2006.

[SS05] W. N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248, 2005.

[ST95] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.