Lazy TSO Reachability

Ahmed Bouajjani[1], Georgel Calin[2], Egor Derevenetc[2,3], and Roland Meyer[2]

[1] LIAFA, University Paris 7
[2] University of Kaiserslautern
[3] Fraunhofer ITWM

Abstract. We address the problem of checking state reachability for programs running under Total Store Order (TSO). The problem has been shown to be decidable but the cost is prohibitive, namely non-primitive recursive. We propose here to give up completeness. Our contribution is a new algorithm for TSO reachability: it uses the standard SC semantics and introduces the TSO semantics lazily and only where needed. At the heart of our algorithm is an iterative refinement of the program of interest. If the program's goal state is SC-reachable, we are done. If the goal state is not SC-reachable, this may be due to the fact that SC under-approximates TSO. We employ a second algorithm that determines TSO computations which are infeasible under SC, and hence likely to lead to new states. We enrich the program to emulate, under SC, these TSO computations. Altogether, this yields an iterative under-approximation that we prove sound and complete for bug hunting, i.e., a semi-decision procedure halting for positive cases of reachability. We have implemented the procedure as an extension to the tool Trencher [1] and compared it to the Memorax [2] and CBMC [14] model checkers.

1 Introduction

Sequential consistency (SC) [19] is the semantics typically assumed for parallel programs. Under SC, instructions are executed atomically and in program order. When programs are executed on an Intel x86 processor, however, they are only guaranteed a weaker semantics known as Total Store Order (TSO). TSO weakens the synchronization guarantees given by SC, which in turn may lead to erroneous behavior. TSO reflects the architectural optimization of store buffers. To reduce the latency of memory accesses, store commands are added to a thread-local FIFO buffer and only later executed on memory.

To check for correct behavior, reachability techniques have proven useful. Given a program and a goal state, the task is to check whether the state is reachable. To give an example, assertion failures can be phrased as reachability problems. Reachability depends on the underlying semantics. Under SC, the problem is known to be PSpace-complete [16]. Under TSO, it is considerably more difficult: although decidable, it is non-primitive recursive-hard [8].

Owing to the high complexity of TSO reachability, tools rarely provide decision procedures [2]. Instead, most approaches implement approximations. Typical approximations of TSO reachability bound the number of loop iterations [5, 6], the number of context switches between threads [9], or the size of

© Springer-Verlag Berlin Heidelberg 2015
A. Egyed and I. Schaefer (Eds.): FASE 2015, LNCS 9033, pp. 267–282, 2015.
DOI: 10.1007/978-3-662-46675-9_18

store buffers [17, 18]. What all these approaches have in common is that they introduce store buffering in the *whole* program. We claim that such a comprehensive instrumentation is unnecessarily heavy.

The idea of our method is to introduce store buffering lazily and only where needed. Unlike [2], we do not target completeness. Instead, we argue that our lazy TSO reachability checker is useful for a fast detection of bugs that are due to the TSO semantics. At a high level, we solve the expensive TSO reachability problem with a series of cheap SC reachability checks — very much like SAT solvers are invoked as subroutines of costlier analyses. The SC checks run interleaved with queries to an oracle. The task of the oracle is to suggest sequences of instructions that should be considered under TSO, which means they are likely to lead to TSO-reachable states outside SC.

To be more precise, the algorithm iteratively repeats the following steps. First, it checks whether the goal state is SC-reachable. If this is the case, the state will be TSO-reachable as well and the algorithm returns. If the state is not SC-reachable, the algorithm asks the oracle for a sequence of instructions and encodes the TSO behavior of the sequence into the input program. As a result, precisely this TSO behavior becomes available under SC. The encoding is linear in the size of the input program and in the length of the sequence.

The algorithm is a semi-decision procedure: it always returns correct answers and is guaranteed to terminate if the goal state is TSO-reachable. This guarantee relies on one assumption on the oracle. If the oracle returns the empty sequence, then the SC- and the TSO-reachable states of the input program have to coincide. We also come up with a good oracle: robustness checkers naturally meet the above requirement. Intuitively, a program is robust against TSO if its partial order-behaviors (reflecting data and control dependencies) under TSO and under SC coincide. Robustness is much easier than TSO reachability, actually PSPACE-complete [10, 11], and hence well-suited for iterative invocations.

We have implemented lazy TSO reachability as an extension to our tool TRENCHER [1], reusing the robustness checking algorithms of TRENCHER to derive an oracle. The implementation is able to solve positive instances of TSO reachability as well as correctly determine safety for robust programs. The source code and experiments are available online [1].

The structure of the paper is as follows. We introduce parallel programs with their TSO and their SC semantics in Section 2. Section 3 presents our main contribution, the lazy approach to solving TSO reachability. Section 4 describes the robustness-based oracle. The experimental evaluation is given in Section 5.

Related Work

As already mentioned, TSO reachability was proven decidable but non-primitive recursive [8] in the case of a finite number of threads and a finite data domain. In the same setting, robustness was shown to be PSPACE-complete [11]. Checking and enforcing robustness against weak memory models has been addressed in [3, 7, 10–13, 24]. The first work to give an efficient sound and complete decision procedure for checking robustness is [10].

The works [2,21,22] propose state-based techniques to solve TSO reachability. An under-approximative method that uses bounded context switching is given in [9]. It encodes store buffers into a linear-size instrumentation, and the instrumented program is checked for SC reachability. The under-approximative techniques of [5,6] are able to guarantee safety only for programs with bounded loops. On the other side of the spectrum, over-approximative analyses abstract store buffers into sets combined with bounded queues [17,18].

2 Parallel Programs

We use automata to define the syntax and the semantics of parallel programs. A (non-deterministic) *automaton* over an alphabet Σ is a tuple $A = (\Sigma, S, \rightarrow, s_0)$, where S is a set of states, $\rightarrow \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times S$ is a set of transitions, and $s_0 \in S$ is an initial state. The automaton is *finite* if the transition relation $\rightarrow$ is finite. We write $s \xrightarrow{a} s'$ if $(s, a, s') \in \rightarrow$, and extend the transition relation to sequences $w \in \Sigma^*$ as expected. The *language of A with final states $F \subseteq S$* is $\mathcal{L}_F(A) := \{w \in \Sigma^* \mid s_0 \xrightarrow{w} s \in F\}$. We say that state $s \in S$ is *reachable* if $s_0 \xrightarrow{w} s$ for some sequence $w \in \Sigma^*$. Letter *a precedes b in w*, denoted by $a <_w b$, if $w = w_1 \cdot a \cdot w_2 \cdot b \cdot w_3$ for some $w_1, w_2, w_3 \in \Sigma^*$.

A parallel program P is a finite sequence of threads that are identified by indices t from TID. Each thread $t := (Com_t, Q_t, I_t, q_{0,t})$ is a finite automaton with transitions I_t that we call *instructions*. Instructions I_t are labelled by *commands* from the set Com_t which we define in the next paragraph. We assume, wlog., that states of different threads are disjoint. This implies that the sets of instructions of different threads are distinct. We use $I := \biguplus_{t \in \mathsf{TID}} I_t$ for all instructions and $Com := \bigcup_{t \in \mathsf{TID}} Com_t$ for all commands. For an instruction $inst := (s, cmd, s')$ in I, we define $cmd(inst) := cmd$, $src(inst) := s$, and $dst(inst) := s'$.

To define the set of commands, let DOM be a finite domain of values that we also use as addresses. We assume that value 0 is in DOM. For each thread t, let REG_t be a finite set of registers that take their values from DOM. We assume per-thread disjoint sets of registers. The set of expressions of thread t, denoted by EXP_t, is defined over registers from REG_t, constants from DOM, and (unspecified) operators over DOM. If $r \in \mathsf{REG}_t$ and $e, e' \in \mathsf{EXP}_t$, the set of commands Com_t consists of loads from memory $r \leftarrow \mathtt{mem}[e]$, stores to memory $\mathtt{mem}[e] \leftarrow e'$, memory fences $\mathtt{mfence}$, assignments $r \leftarrow e$, and conditionals $\mathtt{assume}\ e$. We write $\mathsf{REG} := \biguplus_{t \in \mathsf{TID}} \mathsf{REG}_t$ for all registers and $\mathsf{EXP} := \bigcup_{t \in \mathsf{TID}} \mathsf{EXP}_t$ for all expressions.

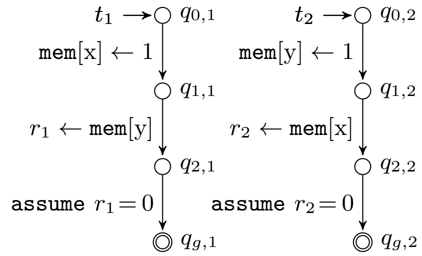


Fig. 1. Simplified Dekker's algorithm

The program in Figure 1 serves as our running example. It consists of two threads t_1 and t_2 implementing a mutual exclusion protocol. Initially, the addresses x and y contain 0. The first thread signals its intent to enter the critical

section by setting variable x to 1. Next, the thread checks whether the second thread wants to enter the critical section, too. It reads variable y and, if it is 0, the first thread enters its critical section. The critical section actually is the state $q_{g,1}$. The second thread behaves symmetrically.

2.1 Semantics of Parallel Programs

The semantics of a parallel program P under memory model $\mathrm{M} = \mathrm{TSO}$ and $\mathrm{M} = \mathrm{SC}$ follows [23]. We define the semantics in terms of a *state-space automaton* $X_{\mathrm{M}}(P) := (E, S_{\mathrm{M}}, \Delta_{\mathrm{M}}, s_0)$. Each state $s = (\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \in S_{\mathrm{M}}$ is a tuple where the program counter $\mathsf{pc}\colon \mathsf{TID} \to Q$ holds the current control state of each thread, the valuation $\mathsf{val}\colon \mathsf{REG} \cup \mathsf{DOM} \to \mathsf{DOM}$ holds the values stored in registers and at memory addresses, and the buffer configuration $\mathsf{buf}\colon \mathsf{TID} \to (\mathsf{DOM} \times \mathsf{DOM})^*$ holds a sequence of address-value pairs.

In the *initial state* $s_0 := (\mathsf{pc}_0, \mathsf{val}_0, \mathsf{buf}_0)$, the program counter holds the initial control states, $\mathsf{pc}_0(t) := q_{0,t}$ for all $t \in \mathsf{TID}$, all registers and addresses contain value 0, and all buffers are empty, $\mathsf{buf}_0(t) := \varepsilon$ for all $t \in \mathsf{TID}$.

The transition relation Δ_{TSO} for TSO satisfies the rules given in Figure 2. There are two more rules for register assignments and conditionals that are standard and omitted. TSO architectures implement (FIFO) store buffering, which means stores are buffered for later execution on the shared memory. Loads from an address a take their value from the most recent store to address a that is buffered. If there is no such buffered store, they access the main memory. This is modelled by the Rules (LB) and (LM). Rule (ST) enqueues store operations as address-value pairs to the buffer. Rule (MEM) non-deterministically dequeues store operations and executes them on memory. Rule (F) states that a thread can execute a fence only if its buffer is empty. As can be seen from Figure 2, events labelling TSO transitions take the form $E \subseteq \mathsf{TID} \times (I \cup \{\text{flush}\}) \times (\mathsf{DOM} \cup \{\bot\})$.

The SC [19] semantics is simpler than TSO in that stores are not buffered. Technically, we keep the set of states but change the transitions so that Rule (ST) is immediately followed by Rule (MEM).

We are interested in the *computations* of program P under $\mathrm{M} \in \{\mathrm{TSO}, \mathrm{SC}\}$. They are given by $\mathcal{C}_{\mathrm{M}}(P) := \mathcal{L}_F(X_{\mathrm{M}}(P))$, where F is the set of states with empty buffers. With this choice of final states, we avoid incomplete computations that have pending stores. Note that all SC states have empty buffers, which means the SC computations form a subset of the TSO computations: $\mathcal{C}_{\mathrm{SC}}(P) \subseteq \mathcal{C}_{\mathrm{TSO}}(P)$. We will use notation $\mathit{Reach}_{\mathrm{M}}(P)$ for the set of all states $s \in F$ that are reachable by some computation in $\mathcal{C}_{\mathrm{M}}(P)$.

To give an example, the program from Figure 1 admits the TSO computation τ_{wit} below where the store of the first thread is flushed at the end:

$$\tau_{\mathrm{wit}} \;=\; \mathtt{store}_1 \cdot \mathtt{load}_1 \cdot \mathtt{store}_2 \cdot \mathtt{flush}_2 \cdot \mathtt{load}_2 \cdot \mathtt{flush}_1.$$

Consider an event $\mathsf{e} = (t, \mathit{inst}, a)$. By $\mathit{thread}(\mathsf{e}) := t$ we refer to the thread that produced the event. Function $\mathit{inst}(\mathsf{e}) := \mathit{inst}$ returns the instruction. For flush events, $\mathit{inst}(\mathsf{e})$ gives the instruction of the matching store event. By $\mathit{addr}(\mathsf{e}) := a$

$$\frac{cmd = r \leftarrow \texttt{mem}[e_a] \quad \mathsf{buf}(t) \downarrow (\{a\} \times \mathsf{DOM}) = (a, v) \cdot \beta}{s \xrightarrow{(t, inst, a)} (\mathsf{pc}', \mathsf{val}[r := v], \mathsf{buf})} \text{(LB)}$$

$$\frac{cmd = r \leftarrow \texttt{mem}[e_a] \quad \mathsf{buf}(t) \downarrow (\{a\} \times \mathsf{DOM}) = \varepsilon}{s \xrightarrow{(t, inst, a)} (\mathsf{pc}', \mathsf{val}[r := \mathsf{val}(a)], \mathsf{buf})} \text{(LM)}$$

$$\frac{cmd = \texttt{mem}[e_a] \leftarrow e_v}{s \xrightarrow{(t, inst, a)} (\mathsf{pc}', \mathsf{val}, \mathsf{buf}[t := (a, v) \cdot \mathsf{buf}(t)])} \text{(ST)}$$

$$\frac{\mathsf{buf}(t) = \beta \cdot (a, v)}{s \xrightarrow{(t, \text{flush}, a)} (\mathsf{pc}, \mathsf{val}[a := v], \mathsf{buf}[t := \beta])} \text{(MEM)} \qquad \frac{cmd = \texttt{mfence} \quad \mathsf{buf}(t) = \varepsilon}{s \xrightarrow{(t, inst, \bot)} (\mathsf{pc}', \mathsf{val}, \mathsf{buf})} \text{(F)}$$

Fig. 2. Transition rules for $X_{\text{TSO}}(P)$ assuming $s = (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ with $\mathsf{pc}(t) = q$ and $inst = (q, cmd, q')$ in thread t. The program counter is always set to $\mathsf{pc}' = \mathsf{pc}[t := q']$. We assume $a = \widehat{e_a}$ to be the address returned by an address expression e_a and $v = \widehat{e_v}$ the value returned by a value expression e_v. We use $\mathsf{buf}(t) \downarrow (\{a\} \times \mathsf{DOM})$ to project the buffer content $\mathsf{buf}(t)$ to store operations that access address a.

we denote the address that is accessed (if any). In the example, $thread(\texttt{store}_1) = t_1$, $inst(\texttt{store}_1) = q_{0,1} \xrightarrow{\texttt{mem}[\text{x}] \leftarrow 1} q_{1,1}$, and $addr(\texttt{store}_1) = \text{x}$.

3 Lazy TSO Reachability

We introduce the reachability problem and present our main contribution: an algorithm that checks TSO reachability lazily. The iterative algorithm queries an oracle to identify sequences of instructions that, under the TSO semantics, lead to states not reachable under SC. In Section 3.1, we show that the algorithm yields a sound and complete semi-decision procedure.

Given a memory model $\text{M} \in \{\text{SC}, \text{TSO}\}$, the M reachability problem expects as input a program P and a set of *goal states* $G \subseteq S_{\text{M}}$. We are mostly interested in the control state of each thread. Therefore, goal states $(\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ typically specify a program counter pc but leave the memory valuation unconstrained. Formally, the *M reachability problem* asks if some state in G is reachable in the automaton $X_{\text{M}}(P)$.

Given: A parallel program P and goal states G.
Problem: Decide $\mathcal{L}_{F \cap G}(X_{\text{M}}(P)) \neq \emptyset$.

We use notation $Reach_{\text{M}}(P) \cap G$ for the set of reachable final goal states in P.

Instead of solving reachability under TSO directly, the algorithm we propose solves SC reachability and, if no goal state is reachable, tries to lazily introduce store buffering on a certain control path of the program. The algorithm delegates choosing the control path to an *oracle function* $\mathcal{O}$. Given an input program R, the oracle returns a sequence of instructions I^* in that program. Formally, the oracle satisfies the following requirements:

- If $\mathcal{O}(R) = \varepsilon$ then $Reach_{\mathrm{SC}}(R) = Reach_{\mathrm{TSO}}(R)$.
- Otherwise, $\mathcal{O}(R) = inst_1 inst_2 \ldots inst_n$ with $cmd(inst_1)$ a store, $cmd(inst_n)$ a load, $cmd(inst_i) \neq \texttt{mfence}$, and $dst(inst_i) = src(inst_{i+1})$ for $i \in [1..n-1]$.

The lazy TSO reachability checker is outlined in Algorithm 1. As input, it takes a program P and an oracle $\mathcal{O}$. We assume some control states in each thread to be marked to define a set of goal states. The algorithm returns *true* iff the program can reach a goal state under TSO. It works as follows. First, it creates a copy R of the program P. Next, it checks if a goal state is SC-reachable in R (Line 3). If that is the case, the algorithm returns *true*. Otherwise, it asks the oracle $\mathcal{O}$ where in the program to introduce store buffering. If $\mathcal{O}(R) \neq \varepsilon$, the algorithm extends R to emulate store buffering on the path $\mathcal{O}(R)$ under SC (Line 8). Then it goes back to the beginning of the loop. If $\mathcal{O}(R) = \varepsilon$, by the first property of oracles, R has the same reachable states under SC and under TSO. This means the algorithm can safely return *false* (Line 10). Note that, since R emulates TSO behavior of P, the algorithm solves TSO reachability for P.

Algorithm 1. Lazy TSO Reachability Checker

Input: Marked program P and oracle $\mathcal{O}$
Output: *true* if some goal state is TSO-reachable in P
false if no goal state is TSO-reachable in P

```
1:  R := P;
2:  while true do
3:    if Reach_SC(P) ∩ G ≠ ∅ then     {check if some goal state is SC-reachable}
4:      return true;
5:    else
6:      σ := O(R);                    {ask the oracle where to use store buffering}
7:      if σ ≠ ε then
8:        R := R ⊕ σ;
9:      else
10:       return false;
```

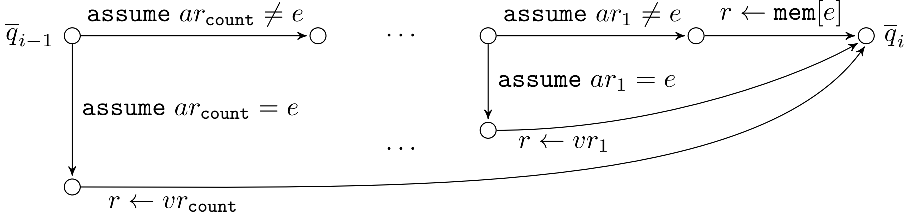
Let $\sigma := \mathcal{O}(R) = inst_1 inst_2 \ldots inst_n$ and let $t := (Com_t, Q_t, I_t, q_{0,t})$ be the thread of the instructions in σ. The modified program $R \oplus \sigma$ replaces t by a new thread $t \oplus \sigma$. The new thread emulates under SC the TSO semantics of σ. Formally, the *extension of t by σ* is $t \oplus \sigma := (Com'_t, Q'_t, I'_t, q_{0,t})$. The thread is obtained from t by adding sequences of instructions starting from $\overline{q}_0 := src(inst_1)$. To remember the addresses and values of the buffered stores, we use auxiliary registers $ar_1, \ldots, ar_{\texttt{max}}$ and $vr_1, \ldots, vr_{\texttt{max}}$, where $\texttt{max} \leq n-1$ is the total number of store instructions in σ. The sets $Com'_t \supseteq Com_t$ and $Q'_t \supseteq Q_t$ are extended as necessary.

We define the extension by describing the new transitions that are added to I'_t for each $inst_i$. In our construction, we use a variable `count` to keep track

of the number of store instructions already processed. Initially, $Q'_t := Q_t$ and $\texttt{count} := 0$. Based on the type of instructions, we distinguish the following cases.

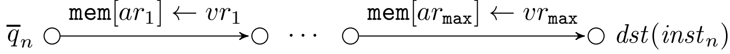
If $cmd(inst_i) = \texttt{mem}[e] \leftarrow e'$, we increment count by 1 and add instructions that remember the address and the value being written in $ar_{\texttt{count}}$ and $vr_{\texttt{count}}$.

If $cmd(inst_i) = r \leftarrow \texttt{mem}[e]$, we add instructions to I'_t that perform a load from memory only when a load from the simulated buffer is not possible. More precisely, if $j \in [1, \texttt{count}]$ is found so that $ar_j = e$, register r is assigned the value of vr_j. Otherwise, r receives its value from the address indicated by e.

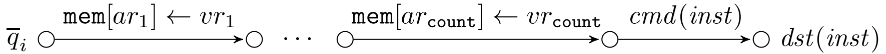


If $cmd(inst_i)$ is an assignment or a conditional, we add $(\overline{q}_{i-1}, cmd(inst_i), \overline{q}_i)$ to I'_t. By the definition of an oracle, $cmd(inst_i)$ is never a fence.

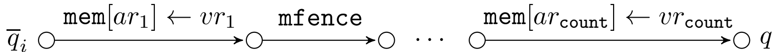
The above cases handle all instructions in σ. So far, the extension added new instructions to I'_t that lead through the fresh states $\overline{q}_1, \ldots, \overline{q}_n$. Out of control state $\overline{q}_n$, we now recreate the sequence of stores remembered by the auxiliary registers. Then we return to the control flow of the original thread t.



Next, we remove $inst_1$ from the program. This prevents the oracle from discovering in the future another instruction sequence that is essentially the same as σ. As we will show, this is key to guaranteeing termination of the algorithm for acyclic programs. However, the removal of $inst_1$ may reduce the set of TSO-reachable states. To overcome this problem, we insert additional instructions. Consider an instruction $inst \in I_t$ with $src(inst) = src(inst_i)$ for some $i \in [1..n]$ and assume that $inst \neq inst_i$. We add instructions that recreate the stores buffered in the auxiliary registers and return to $dst(inst)$.



Similarly, for all load instructions $inst_i$ as well as out of $\overline{q}_1$ we add instructions that flush and fence the pair (ar_1, vr_1), make visible the remaining buffered stores, and return to state q in the original control flow. Below, $q := src(inst_i)$ if $inst_i$ is a load and $q := dst(inst_1)$, otherwise. Intuitively, this captures behaviors that delay $inst_1$ past loads earlier than $inst_n$, and that do not delay $inst_1$ past the first load in σ.



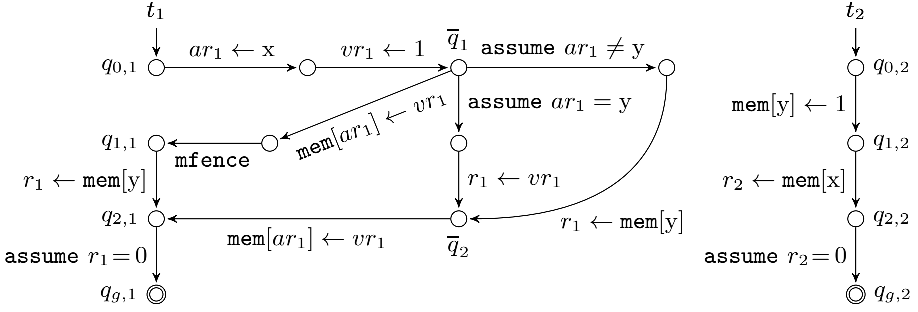


Fig. 3. Extension by $\mathit{inst}(\mathtt{store}_1) \cdot \mathit{inst}(\mathtt{load}_1)$ of the program in Figure 1. Goal state $(\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ with $\mathsf{val}(\mathrm{x}) = \mathsf{val}(\mathrm{y}) = 1$ and $\mathsf{val}(r_1) = \mathsf{val}(r_2) = 0$ is now SC-reachable.

Figure 3 shows the extension of the program in Figure 1 by the instruction sequence $\mathit{inst}(\mathtt{store}_1) \cdot \mathit{inst}(\mathtt{load}_1) := q_{0,1} \xrightarrow{\mathtt{mem}[\mathrm{x}] \leftarrow 1} q_{1,1} \xrightarrow{r_1 \leftarrow \mathtt{mem}[\mathrm{y}]} q_{1,2}$.

3.1 Soundness and Completeness

We show that Algorithm 1 is a decision procedure for acyclic programs. From here until (inclusively) Theorem 3 we assume that all programs are acyclic, i.e., their instructions and control states form directed acyclic graphs. Theorem 4 then explains how Algorithm 1 yields a semi-decision procedure for all programs.

We first prove the extension sound and complete (Lemma 1): extending R by sequence $\sigma := \mathcal{O}(R)$ does neither add nor remove TSO-reachable states. Afterwards, Lemma 2 shows that if Algorithm 1 extends R by σ (Line 8) then, in subsequent iterations of the algorithm, no new sequence returned by the oracle is the same as σ (projected back to P). Next, by the first condition of an oracle and using Lemma 2, we establish that Algorithm 1 is a decision procedure for acyclic programs (Theorem 3). Finally, we show that Algorithm 1 can be turned into a semi-decision procedure for all programs using a bounded model checking approach (Theorem 4).

Lemma 1. *Let $\mathit{DOM} \cup \mathit{REG}$ be the addresses and registers of program R and let $\sigma := \mathcal{O}(R)$. Then we have $(\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \in \mathit{Reach}_{TSO}(R)$ if and only if $(\mathsf{pc}, \mathsf{val}', \mathsf{buf}) \in \mathit{Reach}_{TSO}(R \oplus \sigma)$ with $\mathsf{val}(a) = \mathsf{val}'(a)$ for all $a \in \mathit{DOM} \cup \mathit{REG}$.*

Let t be the thread that differs in R and $R \oplus \sigma$. To prove Lemma 1, one can show that for any prefix α' of $\alpha \in \mathcal{C}_{\mathrm{TSO}}(R)$ there is a prefix β' of $\beta \in \mathcal{C}_{\mathrm{TSO}}(R \oplus \sigma)$, and vice versa, that maintain the following invariants.

Inv-0 $s_0 \xrightarrow{\alpha'} (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ and $s_0 \xrightarrow{\beta'} (\mathsf{pc}', \mathsf{val}', \mathsf{buf}')$.

Inv-1 If pc and pc' differ, they only differ for thread t. If $\mathsf{pc}(t) \neq \mathsf{pc}'(t)$, then $\mathsf{pc}(t) = \mathit{dst}(\mathit{inst}_i)$ and $\mathsf{pc}'(t) = \overline{q}_i$ for some $i \in [1..n-1]$.

Inv-2 $\mathsf{val}'(a) = \mathsf{val}(a)$ for all $a \in \mathsf{DOM} \cup \mathsf{REG}$.

Inv-3 buf and buf' differ at most for t. If $\mathsf{buf}(t) \neq \mathsf{buf}'(t)$, then $\mathsf{pc}'(t) = \overline{q}_i$ for some $i \in [1..n-1]$ and $\mathsf{buf}(t) = (\widehat{ar}_{\mathtt{count}}, \widehat{vr}_{\mathtt{count}}) \cdots (\widehat{ar}_1, \widehat{vr}_1) \cdot \mathsf{buf}'(t)$ where $\mathtt{count}$ stores are seen along σ from $\mathit{src}(\mathit{inst}_1)$ to $\mathit{dst}(\mathit{inst}_i)$.

We now show that the oracle never suggests the same sequence σ twice. Since in $R \oplus \sigma$ we introduce new instructions that correspond to instructions in R, we have to map back sequences of instructions $I_\oplus$ in $R \oplus \sigma$ to sequences of instructions I in R. Intuitively, the mapping gives the original instructions from which the sequence was produced. Formally, we define a family of projection functions $h_\sigma \colon I_\oplus^* \to I^*$ with $h_\sigma(\varepsilon) := \varepsilon$ and $h_\sigma(w \cdot \mathit{inst}) := h_\sigma(w) \cdot h_\sigma(\mathit{inst})$. For an instruction $\mathit{inst} \in I_\oplus$, we define $h_\sigma(\mathit{inst}) := \mathit{inst}$ provided $\mathit{inst} \in I$. We set $h_\sigma(\mathit{inst}) := \mathit{inst}_i$ if inst is a first instruction on the path between $\overline{q}_{i-1}$ and $\overline{q}_i$ for some $i \in [1..n]$. In all other cases, we delete the instruction, $h_\sigma(\mathit{inst}) := \varepsilon$. Then, if $R_0 := P$ is the original program, σ_j is the sequence that the oracle returns in iteration $j \in \mathbb{N}$ of the while loop, and w is a sequence of instructions in R_{j+1}, we define $h(w) := h_{\sigma_0}(\dots h_{\sigma_j}(w))$. This latter function maps sequences of instructions in program R_{j+1} back to sequences of instructions in P.

We are ready to state our key lemma. Intuitively, if the oracle in Algorithm 1 returns $\sigma := \mathcal{O}(R)$ and $\sigma' := \mathcal{O}(R \oplus \sigma)$ then, necessarily, $h(\sigma') \neq h(\sigma)$.

Lemma 2. *Let $R_0 := P$ and $R_{i+1} := R_i \oplus \sigma_i$ for $\sigma_i := \mathcal{O}(R_i)$ as in Algorithm 1. If $\sigma_{j+1} \neq \varepsilon$ then $h(\sigma_{j+1}) \neq h(\sigma_i)$ for all $i \leq j$.*

We can now prove Algorithm 1 sound and complete for acyclic programs (Theorem 3). Lemma 2 and the assumption that the input program is acyclic ensure that if no goal state is found SC-reachable (Line 4), then Algorithm 1 eventually runs out of sequences σ to return (Line 7). If that is the case, $\mathcal{O}(R)$ returns ε in the last iteration of Algorithm 1. By the first oracle condition, we know that the SC- and TSO-reachable states of R are the same. Hence, no goal state is TSO-reachable in R and, by Lemma 1, no goal state is TSO-reachable in the input program P either. Otherwise, a goal state s is SC-reachable by some computation τ in R_j for some $j \in \mathbb{N}$ and, by Lemma 1, there is a TSO computation in P corresponding to τ that reaches s.

Theorem 3. *For acyclic programs, Algorithm 1 terminates. Moreover, it returns true on input P if and only if $\mathit{Reach}_{TSO}(P) \cap G \neq \emptyset$.*

To establish that Algorithm 1 is a semi-decision procedure for all programs, one can use an iterative bounded model checking approach. Bounded model checking unrolls the input program P up to a bound $k \in \mathbb{N}$ on the length of computations. Then Algorithm 1 is applied to the resulting programs P_k. If it finds a goal state TSO-reachable in P_k, this state corresponds to a TSO-reachable goal state in P. Otherwise, we increase k and try again. By Theorem 3, we know that Algorithm 1 is a decision procedure for each P_k. This implies that Algorithm 1 together with iterative bounded model checking yields a semi-decision procedure that terminates for all positive instances of TSO reachability. For negative instances of TSO reachability, however, the procedure is guaranteed to terminate only if the input program P is acyclic.

Theorem 4. *We have $G \cap \mathit{Reach}_{TSO}(P) \neq \emptyset$ if and only if, for large enough $k \in \mathbb{N}$, Algorithm 1 returns true on input P_k.*

4 A Robustness-Based Oracle

This section argues why robustness yields an oracle. Robustness [7,10,13,24] is a correctness criterion requiring that for each TSO computation of a program there is an SC computation that has the same data and control dependencies. Delays due to store buffering are still allowed, as long as they do not produce dependencies between instructions that SC computations forbid.

Dependencies between events are described in terms of the *happens-before* relation of a computation $\tau \in \mathcal{C}_{\text{TSO}}(P)$. The happens-before relation is a union of the three relations that we define below: $\rightarrow_{hb}(\tau) := \rightarrow_{po} \cup \leftrightarrow \cup \rightarrow_{cf}$.

The *program order relation* $\rightarrow_{po}$ is the order in which threads issue their commands. Formally, it is the union of the program order relations for all threads: $\rightarrow_{po} := \bigcup_{t \in \mathsf{TID}} \rightarrow_{po}^{t}$. Let τ' be the subsequence of all non-flush events of thread t in τ. Then $\rightarrow_{po}^{t} := <_{\tau'}$.

The *equivalence relation* $\leftrightarrow$ links, in each thread, flush events and their matching store events: $(t, inst, a) \leftrightarrow (t, \text{flush}, a)$.

The *conflict relation* $\rightarrow_{cf}$ orders accesses to the same address. Assume, on the one hand, that $\tau = \tau_1 \cdot \texttt{store} \cdot \tau_2 \cdot \texttt{load} \cdot \tau_3 \cdot \texttt{flush} \cdot \tau_4$ such that $\texttt{store} \leftrightarrow \texttt{flush}$, events `store` and `load` access the same address a and come from thread t, and there is no other store event $\texttt{store}' \in \tau_2$ such that $\mathit{thread}(\texttt{store}') = t$ and $\mathit{addr}(\texttt{store}') = a$. Then the load event `load` is an *early read* of the value buffered by the event `store` and $\texttt{store} \rightarrow_{cf} \texttt{load}$.

On the other hand, assume $\tau = \tau_1 \cdot \texttt{e} \cdot \tau_2 \cdot \texttt{e}' \cdot \tau_3$ such that $\texttt{e}$ and $\texttt{e}'$ are either load or flush events that access the same address a, neither $\texttt{e}$ nor $\texttt{e}'$ is an early read, and at least one of $\texttt{e}$ or $\texttt{e}'$ is a flush to a. If there is no other flush event $\texttt{flush} \in \tau_2$ with $\mathit{addr}(\texttt{flush}) = a$ then $\texttt{e} \rightarrow_{cf} \texttt{e}'$.

Figure 4 depicts the happens-before relation of computation τ_{wit}.

A program P is said to be *robust* against TSO if for each computation $\tau \in \mathcal{C}_{\text{TSO}}(P)$ there exists a computation $\tau' \in \mathcal{C}_{\text{SC}}(P)$ such that $\rightarrow_{hb}(\tau) = \rightarrow_{hb}(\tau')$. If a program P is robust, then it reaches the same set of final states under SC and under TSO:

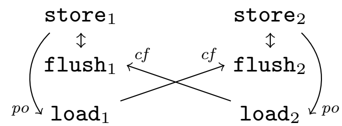


Fig. 4. The relation $\rightarrow_{hb}(\tau_{\text{wit}})$

Lemma 5. *If P is robust against TSO, then $\mathit{Reach}_{SC}(P) = \mathit{Reach}_{TSO}(P)$.*

Our robustness-based oracle makes use of the following characterization of robustness from earlier work [10]: a program P is not robust against TSO iff $\mathcal{C}_{\text{TSO}}(P)$ contains a computation, called *witness*, as in Figure 5.

Lemma 6 ([10]). *Program P is robust against TSO if and only if the set of TSO computations $\mathcal{C}_{TSO}(P)$ contains no witness.*

A witness τ delays stores of only one thread in P. The other threads adhere to the SC semantics. Conditions (W1) – (W4) in Figure 5 describe formally this restrictive behavior. Furthermore, condition (W5) implies that no computation $\tau' \in \mathcal{C}_{\text{SC}}(P)$ can satisfy $\rightarrow_{hb}(\tau) = \rightarrow_{hb}(\tau')$.

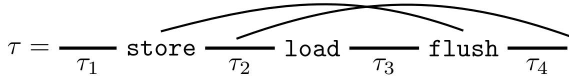


Fig. 5. Witness τ with $\texttt{store} \leftrightarrow \texttt{flush}$ and thread $t := \mathit{thread}(\texttt{store}) = \mathit{thread}(\texttt{load})$. Witnesses satisfy the following constraints: (W1) Only thread t delays stores. (W2) Event $\texttt{flush}$ is the first delayed store of t and $\texttt{load}$ is the last event of t past which $\texttt{flush}$ is delayed. So τ_2 contains neither flush events nor fences of t. (W3) Sequence τ_3 contains no events of thread t. (W4) Sequence τ_4 consists only of flush events $\texttt{e}$ of thread t. All these events $\texttt{e}$ satisfy $\mathit{addr}(\texttt{e}) \neq \mathit{addr}(\texttt{load})$. (W5) We require $\texttt{load} \rightarrow_{hb}^{+} \texttt{e}$ for all events $\texttt{e}$ in $\tau_3 \cdot \texttt{flush}$.

The computation τ_{wit} is a witness for the program in Figure 1. Indeed, in no SC computation of this program can both loads read the initial values of x and y. Relative to Figure 5, we have $\texttt{store} = \texttt{store}_1$, $\texttt{load} = \texttt{load}_1$, $\texttt{flush} = \texttt{flush}_1$, $\tau_3 = \texttt{store}_2 \cdot \texttt{flush}_2 \cdot \texttt{load}_2$, and $\tau_1 = \tau_2 = \tau_4 = \varepsilon$.

The *robustness-based oracle*, given input P, finds a witness τ as in Figure 5 and returns the sequence of instructions for the events in $\texttt{store} \cdot \tau_2 \cdot \texttt{load}$ that belong to thread t. If no witness exists, it returns ε. By Lemmas 5 and 6, this satisfies the oracle conditions from Section 3. Note that, given a robust program and the robustness-based oracle as inputs, Algorithm 1 returns within the first iteration of the while loop.

5 Experiments

We have implemented our lazy TSO reachability algorithm on top of the tool Trencher [1]. Trencher was initially developed for checking robustness and implements the algorithm for finding witness computations described in [10]. Our implementation reuses that algorithm as a robustness-based oracle. Trencher originally used SPIN [15] as back-end SC reachability checker. Currently, Trencher uses our own implementation of a simple SC model checker. This implementation exploits information about the instruction set for partial-order reduction and for live variable analysis. Moreover, it avoids having to compile the verifier executables (pan) as was the case for SPIN.

We have implemented Algorithm 1 with the following amendments. First, the extension does not delete the store instruction inst_1. This ensures the extended program has a superset of the TSO behaviors of the original program. Second, the extension only adds instructions along $\overline{q}_1, \ldots, \overline{q}_n$. The remaining instructions were added to ensure all behaviors of the original program exist in the extended program, once inst_1 is removed. The resulting algorithm is guaranteed to give correct results for cyclic programs. Of course, it cannot be guaranteed to terminate in general. Finally, our implementation explores extensions due to different instruction sequences in parallel, rather than sequentially.

We compare our prototype implementation against two other model checkers that support TSO semantics: Memorax [2] (revision 4f94ab6) and CBMC [14] (version 4.7). Memorax implements a sound and complete reachability checking procedure by reducing to coverability in a well-structured transition system.

CBMC is an SMT-based bounded model checker for C programs. Consequently, it is sound, but not complete: it is complete only up to a given bound on the number of loop iterations in the input program.

We describe three parameterized tests that we performed — more examples are available online [1]. The first one is Lamport's fast mutex [20] (see left-hand-side of Figure 6) where we varied the number of threads. The modified Dekker in Figure 7 is inspired by the examples of the fence-insertion tool MUSKETEER [4] and adds an "N-branching diamond" (see right-hand-side of Figure 7) to both program threads. Lastly, the program in Figure 8 has stores to address x on a length N loop in thread t_1: since t_1 expects to load the initial y value while t_2 expects to load 1 and then 0 from x, an execution that reaches the goal state goes through the length N loop twice.

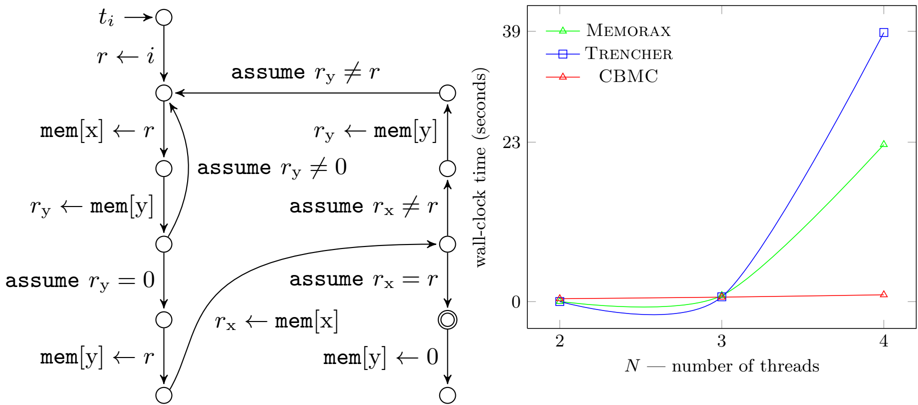


Fig. 6. The i-th Lamport mutex thread (left) and running times for N threads (right)

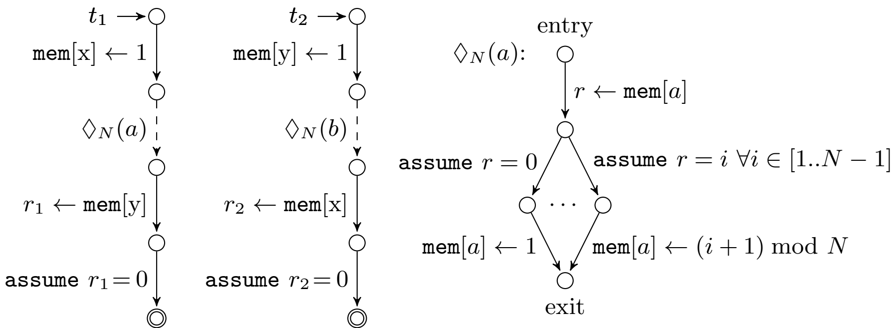


Fig. 7. Dekker's algorithm modified so that an "N-branching diamond" over distinct addresses $a, b \notin \{x, y\}$ is placed between the accesses to x and y. A final goal state is TSO-reachable if the first store is delayed past the last load in either t_1 or t_2.

5.1 Evaluation

We ran all tests on a QEMU @ 2.67GHz virtual machine (16 cores) with 8GB RAM running GNU/Linux. Our comparison does not include robust programs:

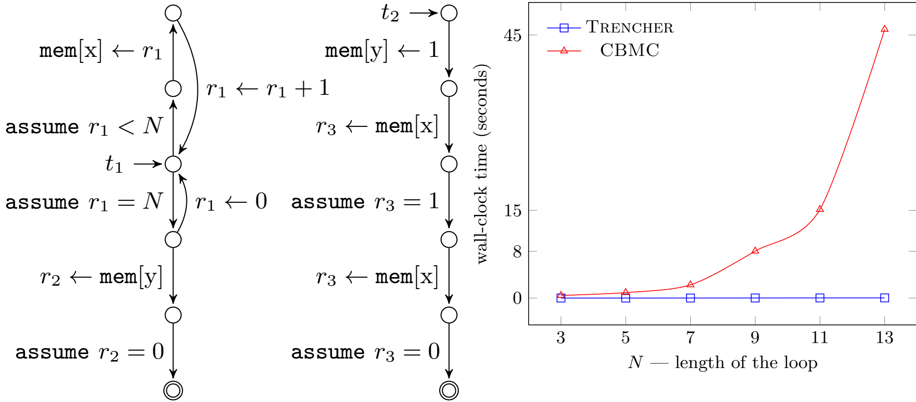


Fig. 8. A final goal state is TSO-reachable if t_1 goes through the (length N) loop two times: once to satisfy `assume` $r_3 = 1$ and the second time to satisfy `assume` $r_3 = 0$

TSO reachability for robust programs can be checked using SC reachability (if SC reachability returns false then one should account for verifying robustness). Moreover, CBMC implements an under-approximative method where the number of loop iterations is bounded. Our robust tests [1], however, contain unbounded loops.

The graph in Figure 6 shows that CBMC will be the fastest to verify Lamport's mutex when increasing the number of threads. This is the case since the smallest unwind bound suffices for CBMC to conclude reachability. For Trencher and Memorax, the considerable difference in verification time when increasing the number of threads is justified by the correlation between the program's data domain size and its thread count. Although not easily noticeable in Figure 6, Memorax's exponential scaling is better than Trencher's: Trencher is slightly faster than Memorax for $N \in \{2, 3\}$ but Memorax outperforms Trencher when $N = 4$. For both Memorax and Trencher our system runs out of memory when $N = 5$.

The graph in Figure 8 shows that our prototype is faster than CBMC for the second parameterized test. Indeed, with increasing N, an ever larger number of constraints need to be generated by CBMC. For Trencher, regardless of the value of N, it takes three SC reachability queries to conclude TSO reachability.

The graph in Figure 9 shows that, for the programs described by Figure 7, our prototype is faster than Memorax. It seems Memorax cannot cope well with the branching factor that the parameter N introduces.

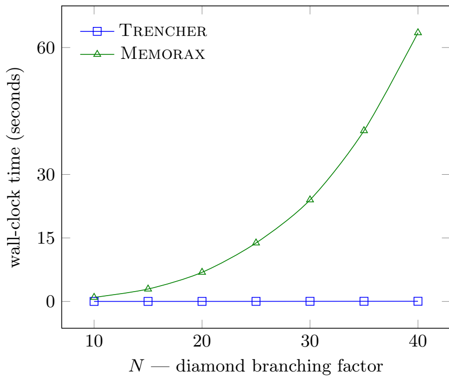


Fig. 9. Running times for Figure 7 tests

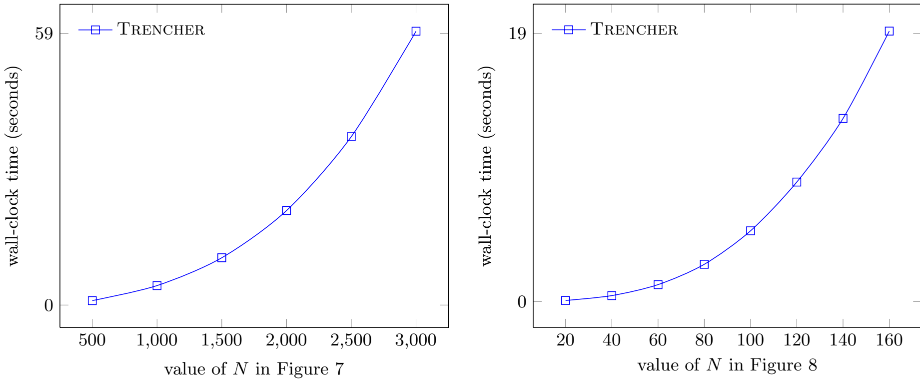


Fig. 10. Additional Trencher results for the programs in Figures 7 and 8. Memorax takes already 1 minute and 24 seconds for the program in Figure 7 and $N = 50$, while CBMC takes 8 minutes and 35 seconds for the program in Figure 8 and $N = 20$.

From Figures 8, 9, and 10 one can infer that Trencher scales better than Memorax and CBMC for the Figure 7 and Figure 8 examples, respectively.

5.2 Discussion

Because we find several witnesses in parallel, throughout the experiments our implementation required up to 2 iterations of the loop in Algorithm 1. In the case of robust programs, one iteration is always sufficient. This suggests that robustness violations are really the critical behaviors leading to TSO reachability.

The experiments indicate that, at least for some programs with a high branching factor, our implementation is faster than Memorax if a useful witness can be found within a small number of iterations of Algorithm 1. Similarly, our prototype is better than CBMC for programs which require a high unwinding bound to make visible TSO behavior reaching a goal state. Although the two programs by which we show this are rather artificial, we expect such characteristics to occur in actual code. Hence, our approach seems to be strong on an orthogonal set of programs. In a portfolio model checker, it could be used as a promising alternative to the existing techniques.

To evaluate the practicality of our method, more experiments are needed. In particular, we hope to be able to substantiate the above conjecture for concrete programs with behavior like that depicted in Figures 7 and 8. Unfortunately, there seems to be no clear way of translating (compiled) C programs into our simplified assembly syntax without substantial abstraction. To handle C code, an alternative would be to reimplement our method within CBMC. But this would force us to determine a-priori a good-enough unwinding bound. Moreover, we could no longer conclude safety of robust programs with unbounded loops.

Acknowledgements. The third author was granted by the Competence Center High Performance Computing and Visualization (CC-HPC) of the Fraunhofer Institute for Industrial Mathematics (ITWM). The work was partially supported by the PROCOPE project ROIS: *Robustness under Realistic Instruction Sets* and by the DFG project R2M2: *Robustness against Relaxed Memory Models*.

References

1. The Trencher tool, `http://concurrency.informatik.uni-kl.de/trencher.html`
2. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counter-Example Guided Fence Insertion under TSO. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 204–219. Springer, Heidelberg (2012)
3. Alglave, J.: A Shared Memory Poetics. PhD thesis, University Paris 7 (2010)
4. Alglave, J., Kroening, D., Nimal, V., Poetzl, D.: Don't Sit on the Fence. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 508–524. Springer, Heidelberg (2014)
5. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software Verification for Weak Memory via Program Transformation. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013)
6. Alglave, J., Kroening, D., Tautschnig, M.: Partial Orders for Efficient BMC of Concurrent Software. CoRR, abs/1301.1629 (2013)
7. Alglave, J., Maranget, L.: Stability in Weak Memory Models. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 50–66. Springer, Heidelberg (2011)
8. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the Verification Problem for Weak Memory Models. In: POPL, pp. 7–18. ACM (2010)
9. Atig, M.F., Bouajjani, A., Parlato, G.: Getting Rid of Store-Buffers in TSO Analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)
10. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and Enforcing Robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013)
11. Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding Robustness against Total Store Ordering. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 428–440. Springer, Heidelberg (2011)
12. Burckhardt, S., Musuvathi, M.: Effective Program Verification for Relaxed Memory Models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
13. Burnim, J., Sen, K., Stergiou, C.: Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 11–25. Springer, Heidelberg (2011)
14. Clarke, E., Kroning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
15. Holzmann, G.J.: The Model Checker SPIN. IEEE Tr. Sof. Eng. 23, 279–295 (1997)
16. Kozen, D.: Lower Bounds for Natural Proof Systems. In: FOCS, pp. 254–266. IEEE Computer Society Press (1977)
17. Kuperstein, M., Vechev, M., Yahav, E.: Partial-Coherence Abstractions for Relaxed Memory Models. In: PLDI, pp. 187–198. ACM (2011)

18. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic Inference of Memory Fences. ACM SIGACT News 43(2), 108–123 (2012)
19. Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Tr. on Com. 28(9), 690–691 (1979)
20. Lamport, L.: A Fast Mutual Exclusion Algorithm. ACM Tr. Com. Sys. 5(1) (1987)
21. Linden, A., Wolper, P.: An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models. In: van de Pol, J., Weber, M. (eds.) Model Checking Software. LNCS, vol. 6349, pp. 212–226. Springer, Heidelberg (2010)
22. Linden, A., Wolper, P.: A Verification-based Approach to Memory Fence Insertion in Relaxed Memory Systems. In: Groce, A., Musuvathi, M. (eds.) SPIN Workshops 2011. LNCS, vol. 6823, pp. 144–160. Springer, Heidelberg (2011)
23. Owens, S., Sarkar, S., Sewell, P.: A Better x86 Memory Model: x86-TSO (extended version). Technical Report CL-TR-745, University of Cambridge (2009)
24. Shasha, D., Snir, M.: Efficient and Correct Execution of Parallel Programs that Share Memory. ACM Tr. on Prog. Lang. and Sys. 10(2), 282–312 (1988)