Reasoning About TSO Programs Using Reduction and Abstraction*

Ahmed Bouajjani[1], Constantin Enea[1], Suha Orhun Mutluergil[2], and Serdar Tasiran[3]

[1] IRIF, University Paris Diderot & CNRS, {abou,cenea}@irif.fr,
[2] Koc University, smutluergil@ku.edu.tr
[3] Amazon Web Services, tasirans@amazon.com

Abstract. We present a method for proving that a program running under the Total Store Ordering (TSO) memory model is robust, i.e., all its TSO computations are equivalent to computations under the Sequential Consistency (SC) semantics. This method is inspired by Lipton's reduction theory for proving atomicity of concurrent programs. For programs which are not robust, we introduce an abstraction mechanism that allows to construct robust programs over-approximating their TSO semantics. This enables the use of proof methods designed for the SC semantics in proving invariants that hold on the TSO semantics of a non-robust program. These techniques have been evaluated on a large set of benchmarks using the infrastructure provided by CIVL, a generic tool for reasoning about concurrent programs under the SC semantics.

1 Introduction

A classical memory model for shared-memory concurrency is Sequential Consistency [16] (SC), where the actions of different threads are interleaved while the program order between actions of each thread is preserved. For performance reasons, modern multiprocessors implement weaker memory models, e.g., Total Store Ordering (TSO) [20] in x86 machines, which relax the program order. For instance, the main feature of TSO is the write-to-read relaxation, which allows reads to overtake writes. This relaxation reflects the fact that writes are buffered before being flushed non-deterministically to the main memory.

Nevertheless, most programmers usually assume that memory accesses happen instantaneously and atomically like in the SC memory model. This assumption is safe for data-race free programs [3]. However, many programs employing lock-free synchronization are not data-race free, e.g., programs implementing synchronization operations and libraries implementing concurrent objects. In most cases, these programs are designed to be robust against relaxations, i.e., they admit the same behaviors as if they were run under SC. Memory fences must

* This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177).

be included appropriately in programs in order to prevent non-SC behaviors. Getting such programs right is a notoriously difficult and error-prone task. Robustness can also be used as a proof method, that allows to reuse the existing SC verification technology. Invariants of a robust program running under SC are also valid for the TSO executions. Therefore, the problem of checking robustness of a program against relaxations of a memory model is important.

In this paper, we address the problem of checking robustness in the case of TSO. We present a methodology for proving robustness which uses the concepts of left/right mover in Lipton's reduction theory [17]. Intuitively, a program statement is a left (resp., right) mover if it commutes to the left (resp., right) with respect to the statements in the other threads. These concepts have been used by Lipton [17] to define a program rewriting technique which enlarges the atomic blocks in a given program while preserving the same set of behaviors. In essence, robustness can also be seen as an atomicity problem: every write statement corresponds to two events, inserting the write into the buffer and flushing the write from the buffer to the main memory, which must be proved to happen atomically, one after the other. However, differently from Lipton's reduction theory, the events that must be proved atomic do not correspond syntactically to different statements in the program. This leads to different uses of these concepts which cannot be seen as a direct instantiation of this theory.

In case programs are not robust, or they cannot be proven robust using our method, we define a program abstraction technique that roughly, makes reads non-deterministic (this follows the idea of combining reduction and abstraction introduced in [12]). The non-determinism added by this abstraction can lead to programs which can be proven robust using our method. Then, any invariant (safety property) of the abstraction, which is valid under the SC semantics, is also valid for the TSO semantics of the original program. As shown in our experiments, this abstraction leads in some cases to programs which reach exactly the same set of configurations as the original program (but these configurations can be reached in different orders), which implies no loss of precision.

We tested the applicability of the proposed reduction and abstraction based techniques on an exhaustive benchmark suite containing 34 challenging programs (from [2] and [7]). These techniques were precise enough for proving robustness of 32 of these programs. One program (presented in Figure 3) is not robust, and required abstraction in order to derive a robust over-approximation. There is only one program which cannot be proved robust using our techniques (although it is robust). We believe however that an extension of our abstraction mechanism to atomic read-write instructions will be able to deal with this case. We leave this question for future work.

An extended version of this paper with missing proofs can be found at [8].

2 Overview

The TSO memory model allows strictly more behaviors than the classic SC memory model: writes are first stored in a thread-local buffer and non-deterministically

```
procedure send(){
    y := r1;
    x := 1;
}
```

```
procedure recv(){
    do{
        r1 := x;
    }while(r1 == 0);
    r2 := y;
}
```

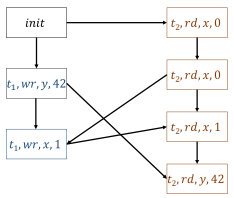

Fig. 1. An example message passing program and a sample trace. Edges of the trace shows the happens before order of global accesses and they are simplified by applying transitive reduction.

flushed into the shared memory at a later time (also, the write buffers are accessed first when reading a shared variable). However, in practice, many programs are *robust*, i.e., they have exactly the same behaviors under TSO and SC. Robustness implies for instance, that any invariant proved under the SC semantics is also an invariant under the TSO semantics. We describe in the following a sound methodology for checking that a program is *robust*, which avoids modeling and verifying TSO behaviors. Moreover, for non-robust programs, we show an abstraction mechanism that allows to obtain robust programs over-approximating the behaviors of the original program.

As a first example, consider the simple "message passing" program in Figure 1. The `send` method sets the value of the "communication" variable y to some predefined value from register $r1$. Then, it raises a flag by setting the variable x to 1. Another thread executes the method `recv` which waits until the flag is set and then, it reads y (and stores the value to register $r2$). This program is robust, TSO doesn't enable new behaviors although the writes may be delayed. For instance, consider the following TSO execution (we assume that $r1 = 42$):

$$\begin{array}{lll} (t_1, isu) & (t_1, isu)(t_1, com, y, 42) & (t_1, com, x, 1) \\ \quad (t_2, rd, x, 0) & (t_2, rd, x, 0) & (t_2, rd, x, 1)(t_2, rd, y, 42) \end{array}$$

The actions of each thread (t_1 or t_2) are aligned horizontally, they are either *issue* actions (isu) for writes being inserted into the local buffer (e.g., the first (t_1, isu) represents the write of y being inserted to the buffer), *commit* actions (com) for writes being flushed to the main memory (e.g., $(t_1, com, y, 42)$ represents the write $y := 42$ being flushed and executed on the shared memory), and *read* actions for reading values of shared variables. Every assignment generates two actions, an issue and a commit. The issue action is "local", it doesn't enable or disable actions of other threads.

The above execution can be "mimicked" by an SC execution. If we had not performed the isu actions of t_1 that early but delayed them until just before their corresponding com actions, we would obtain a valid SC execution of the same program with no need to use store buffers:

$$\begin{array}{lll} & (t_1, wr, y, 42) & (t_1, wr, x, 1) \\ (t_2, rd, x, 0) & (t_2, rd, x, 0) & (t_2, rd, x, 1)(t_2, rd, y, 42) \end{array}$$

Above, consecutive isu and com actions are combined into a single *write* action (wr). This intuition corresponds to an equivalence relation between TSO executions and SC executions: if both executions contain the same actions on the

shared variables (performing the same accesses on the same variables with the same values) and the order of actions on the same variable are the same for both executions, we say that these executions have the same *trace* [21], or that they are *trace-equivalent*. For instance, both the SC and TSO executions given above have the same trace given in Figure 1. The notion of trace is used to formalize robustness for programs running under TSO [7]: a program is called *robust* when every TSO execution has the same trace as an SC execution.

Our method for showing robustness is based on proving that every TSO execution can be permuted to a trace-equivalent SC execution (where issue actions are immediately followed by the corresponding commit actions). We say that an action α moves right until another action β in an execution if we can swap α with every later action until β while preserving the feasibility of the execution (e.g., not invalidating reads and keeping the actions enabled). We observe that if α moves right until β then the execution obtained by moving α just before β has the same trace with the initial execution. We also have the dual notion of moves-left with a similar property. As a corollary, if every issue action moves right until the corresponding commit action or every commit action moves left until the corresponding issue action, we can find an equivalent SC execution. For our execution above, the issue actions of the first thread move right until their corresponding *com* actions. Note that there is a commit action which doesn't move left: moving $(t_1, com, x, 1)$ to the left of $(t_2, rd, x, 0)$ is not possible since it would disable this read.

In general, issue actions and other thread local actions (e.g. statements using local registers only) move right of other threads' actions. Moreover, issue actions (t, isu) move right of commit actions of the same thread that correspond to writes issued before (t, isu). For the message passing program, the issue actions move right until their corresponding commits in all TSO executions since commits cannot be delayed beyond actions of the same thread (for instance reads). Hence, we can safely deduce that the message passing program is robust. However, this reasoning may fail when an assignment is followed by a read of a shared variable in the same thread.

```
procedure foo(){          ||   procedure bar(){
    x := 1;               ||       y := 1;
    r1 := z;              ||       fence
    fence                 ||       r3:= x;
    r2 := y;              ||   }
}
```

Fig. 2. An example store buffering program.

Consider the "store-buffering" like program in Figure 2. This program is also robust. However, the issue action generated by `x := 1` might not not always move right until the corresponding commit. Consider the following execution (we assume that initially, $z = 5$):

$$(t_1, isu) \qquad (t_1, rd, z, 5) \qquad\qquad\qquad (t_1, com, x, 1) \ldots$$
$$(t_2, isu) \qquad\qquad (t_2, com, y, 1)(t_2, \tau)(t_2, rd, x, 0) \qquad \ldots$$

Here, we assumed that t_1 executes `foo` and t_2 executes `bar`. The `fence` instruction generates an action τ. The first issue action of t_1 cannot be moved to the right until the corresponding commit action since this would violate the program order. Moreover, the corresponding commit action does not move left due to the read action of t_2 on x (which would become infeasible).

The key point here is that a later read action by the same thread, $(t_1, rd, z, 5)$, doesn't allow to move the issue action to the right (until the commit). However, this read action moves to the right of other threads actions. So, we can construct an equivalent SC execution by first moving the read action right after the commit $(t_1, com, x, 1)$ and then move the issue action right until the commit action.

In general, we say that an issue (t, isu) of a thread t moves right until the corresponding commit if each read action of t after (t, isu) can move right until the next action of t that follows both the read and the commit. Actually, this property is not required for all such reads. The read actions that follow a fence cannot happen between the issue and the corresponding commit actions. For instance, the last read action of `foo` cannot happen between the first issue of `foo` and its corresponding commit action. Such reads that follow a fence are not required to move right. In addition, we can omit the right-moves check for read actions that read from the thread local buffer (see Section 3 for more details).

In brief, our method for checking robustness does the following for every write instruction (assignment to a shared variable): either the commit action of this write moves left or the actions of later read instructions that come before a fence move right in all executions. This semantic condition can be checked using the concept of movers [18] as follows: every write instruction is either a left-mover or all the read instructions that come before a fence and can be executed later than the write (in an SC execution) are right-movers. Note that this requires no modeling and verification of TSO executions.

For non-robust programs that might reach different configurations under TSO than under SC, we define an abstraction mechanism that replaces read instructions with "non-deterministic" reads that can read more values than the original instructions. The abstracted program has more behaviors than the original one (under both SC and TSO), but it may turn to be robust. When it is robust, we get that any property of its SC semantics holds also for the TSO semantics of the original program.

Consider the work stealing queue implementation in Figure 3. A queue is represented with an array `items`. Its head and tail indices are stored in the shared variables `H` and `T`, respectively. There are three procedures that can operate on this queue: any number of threads may execute the `steal` method and remove an element from the head of the queue, and a single unique thread may execute `put` or `take` methods nondeterministically. The `put` method inserts an element at the tail index and the `take` method removes an element from the tail index.

This program is not robust. If there is a single element in the queue and the `take` method takes it by delaying its writes after some concurrent `steals`, one of the concurrent `steals` might also remove this last element. Popping the same element twice is not possible under SC, but it is possible under TSO semantics. However, we can still prove some properties of this program under TSO. Our robustness check fails on this program because the writes of the worker thread (executing the `put` and `take` methods) are not left movers and the read from the variable H in the `take` method is not a right mover.This read is not a right mover w.r.t. successful CAS actions of the `steal` procedure that increment H.

```
var H,T,items;

procedure steal(){
   local h,t,res;
L1:h := H;
   t := T;
   if(h ≥ t)
     return -1;
   res := items[h];
   if( cas(H,h,h+1) )
     return res;
   else
     goto L1;
}
```

```
procedure put(var elt){
   local t;
   t := T;
   items[t] := elt;
   T := t+1;
}

procedure take(){
   local h,t,res;
L1:t := T;
   T := t-1;
   h := H;     //havoc(h, h ≤ H);
   if( t < h ){
     T := h;
     return -1;
   }
   res := items[t];
   if( t > h )
     return res;
   T := h+1;
   if( cas(H,h,h+1) )
     return task;
   else
     goto L1;
}
```

Fig. 3. Work Stealing Queue.

We apply an abstraction on the instruction of the `take` method that reads from H such that instead of reading the exact value of H, it can read any value less than or equal to the value of H. We write this instruction as $\texttt{havoc}(\texttt{h}, \texttt{h} \leq \texttt{H})$ (it assigns to `h` a nondeterministic value satisfying the constraint $\texttt{h} \leq \texttt{H}$). Note that this abstraction is sound in the sense that it reaches more states under SC/TSO than the original program.

The resulting program is robust. The statement $\texttt{havoc}(\texttt{h}, \texttt{h} \leq \texttt{H})$ is a right mover w.r.t. successful CAS actions of the stealer threads. Hence, for all the write instructions, the reachable read instructions become right movers and our check succeeds. The abstract program satisfies the specification of an idempotent work stealing queue (elements can be dequeued multiple times) which implies that the original program satisfies this specification as well.

3 TSO Robustness

We present the syntax and the semantics of a simple programming language used to state our results. We define both the TSO and the SC semantics, an abstraction of executions called *trace* [21] that intuitively, captures the happens-before relation between actions in an execution, and the notion of robustness.

Syntax. We consider a simple programming language which is defined in Figure 4. Each program $\mathcal{P}$ has a finite number of shared variables $\overrightarrow{x}$ and a finite number of threads ($\overrightarrow{t}$). Also, each thread t_i has a finite set of local registers ($\overrightarrow{r_i}$) and a start label l_i^0. Bodies of the threads are defined as finite sequences of labelled instructions. Each instruction is followed by a `goto` statement which defines the evolution of the program counter. Note that multiple instructions can be assigned to the same label which allows us to write non-deterministic programs and multiple `goto` statements can direct the control to the same label

⟨*prog*⟩ ::= `program` ⟨*pid*⟩ `vars` ⟨*var*⟩* ⟨*thread*⟩*

⟨*thread*⟩ ::= `thread` ⟨*tid*⟩ `regs` ⟨*reg*⟩* `init` ⟨*label*⟩ `begin` ⟨*linst*⟩* `end`

⟨*linst*⟩ ::= ⟨*label*⟩`:` ⟨*inst*⟩`;` `goto` ⟨*label*⟩`;`

⟨*inst*⟩ ::= ⟨*var*⟩ `:=` ⟨*expr*⟩
| ⟨*reg*⟩ `:=` ⟨*expr*⟩
| ⟨*reg*⟩ `:=` ⟨*var*⟩
| `fence`
| ⟨*reg*⟩ `:=` `cas`(⟨*var*⟩, ⟨*expr*⟩, ⟨*expr*⟩)
| `skip`
| `assume` ⟨*bexpr*⟩

Fig. 4. Syntax of the programs. The star (*) indicates zero or more occurrences of the preceding element. ⟨*pid*⟩, ⟨*tid*⟩, ⟨*var*⟩, ⟨*reg*⟩ and ⟨*label*⟩ are elements of their given domains representing the program identifiers, thread identifiers, shared variables, registers and instruction labels, respectively. ⟨*expr*⟩ is an arithmetic expression over ⟨*reg*⟩*. Similarly, ⟨*bexpr*⟩ is a boolean expression over ⟨*reg*⟩*.

which allows us to mimic imperative constructs like loops and conditionals. An assignment to a shared variable ⟨*var*⟩ := ⟨*expr*⟩ is called a *write instruction*. Also, an instruction of the form ⟨*reg*⟩ := ⟨*var*⟩ is called a *read instruction*.

Instructions can read from or write to shared variables or registers. Each instruction accesses at most one shared variable. We assume that the program $\mathcal{P}$ comes with a domain $\mathcal{D}$ of values that are stored in variables and registers, and a set of functions $\mathcal{F}$ used to calculate arithmetic and boolean expressions.

The `fence` statement empties the buffer of the executing thread. The `cas` (compare-and-swap) instruction checks whether the value of its input variable is equal to its second argument. If so, it writes sets third argument as the value of the variable and returns $true$. Otherwise, it returns $false$. In either case, `cas` empties the buffer immediately after it executes. The `assume` statement allows us to check conditions. If the boolean expression it contains holds at that state, it behaves like a `skip`. Otherwise, the execution blocks. Formal description of the instructions are given in Figure 5.

TSO Semantics. Under the TSO memory model, each thread maintains a local queue to buffer write instructions. A state s of the program is a triple of the form (pc, mem, buf). Let $\mathcal{L}$ be the set of available labels in the program $\mathcal{P}$. Then, $pc : \overrightarrow{t} \to \mathcal{L}$ shows the next instruction to be executed for each thread, $mem : \bigcup_{t_i \in \overrightarrow{t}} \overrightarrow{r_i} \cup \overrightarrow{x} \to \mathcal{D}$ represents the current values in shared variables and registers and $buf : \overrightarrow{t} \to (\overrightarrow{x} \times \mathcal{D})^*$ represents the contents of the buffers.

There is a special initial state $s_0 = (pc_0, mem_0, buf_0)$. At the beginning, each thread t_i points to its initial label l_i^0 i.e., $pc_0(t_i) = l_i^0$. We assume that there is a special default value $0 \in \mathcal{D}$. All the shared variables and registers are initiated as 0 i.e., $mem_0(x) = 0$ for all $x \in \bigcup_{t_i \in \overrightarrow{t}} \overrightarrow{r_i} \cup \overrightarrow{x}$. Lastly, all the buffers are initially empty i.e., $buf_0(t_i) = \epsilon$ for all $t_i \in \overrightarrow{t}$.

The transition relation $\to_{TSO}$ between program states is defined in Figure 5. Transitions are labelled by actions. Each action is an element from $\overrightarrow{t} \times (\{\tau, isu\} \cup$

$$\frac{x := ae(\overrightarrow{r_t}) \in ins(pc(t)) \quad v = eval(ae(\overrightarrow{r_t})) \quad x \in \overrightarrow{x}}{(pc, mem, buf) \xrightarrow{(t,isu)}_{TSO} (pc', mem, buf[t \to buf(t) \circ \langle (x, v) \rangle])}$$

$$\frac{buf(t) = \langle (x, v) \rangle \circ buf' \quad x \in \overrightarrow{x}}{(pc, mem, buf) \xrightarrow{(t,com,x,v)}_{TSO} (pc, mem, buf[t \to buf'])}$$

$$\frac{r := ae(\overrightarrow{r_t}) \in ins(pc(t)) \quad v = eval(ae(\overrightarrow{r_t})) \quad r \in \overrightarrow{r_t}}{(pc, mem, buf) \xrightarrow{(t,\tau)}_{TSO} (pc', mem[r \to v], buf)}$$

$$\frac{r := x \in ins(pc(t)) \quad x \in \overrightarrow{x} \quad v = mem(x) \quad x \notin varsOfBuf(buf(t)) \quad r \in \overrightarrow{r_t}}{(pc, mem, buf) \xrightarrow{(t,rd,x,v)}_{TSO} (pc', mem[r \to v], buf)}$$

$$\frac{r := x \in ins(pc(t)) \quad x \in \overrightarrow{x} \quad buf = \alpha \circ \langle (x, v) \rangle \circ \beta \quad x \notin varsOfBuf(\beta) \quad r \in \overrightarrow{r_t}}{(pc, mem, buf) \xrightarrow{(t,rd,x,v)}_{TSO} (pc', mem[r \to v], buf)}$$

$$\frac{\texttt{fence} \in ins(pc(t)) \quad buf(t) = \epsilon}{(pc, mem, buf) \xrightarrow{(t,\tau)}_{TSO} (pc', mem, buf)}$$

$$\frac{r := cas(x, ae_1(\overrightarrow{r_t}), ae_2(\overrightarrow{r_t})) \in ins(pc(t)) \quad mem(x) = eval(ae_1(\overrightarrow{r_t})) \quad buf(t) = \epsilon \quad v = eval(ae_2(\overrightarrow{r_t}))}{(pc, mem, buf) \xrightarrow{(t,isu)(t,com,x,v)}_{TSO} (pc', mem[r \to 1][x \to v], buf)}$$

$$\frac{r := cas(x, ae_1(\overrightarrow{r_t}), ae_2(\overrightarrow{r_t})) \in ins(pc(t)) \quad mem(x) \neq eval(ae_1(\overrightarrow{r_t})) \quad buf(t) = \epsilon \quad v = mem(x)}{(pc, mem, buf) \xrightarrow{(t,rd,x,v)}_{TSO} (pc', mem[r \to 0], buf)}$$

$$\frac{\texttt{assume}\ be(\overrightarrow{r_t}) \in ins(pc(t)) \quad eval(be(\overrightarrow{r_t})) = \top}{(pc, mem, buf) \xrightarrow{(t,\tau)}_{TSO} (pc', mem, buf)}$$

Fig. 5. The TSO Transition Relation. The function ins takes a label l and returns the set of instructions labeled by l. We always assume that $x \in \overrightarrow{x}$, $r \in \overrightarrow{r_t}$ and $pc' = pc[t \to l']$ where $pc(t) : inst\ goto\ l'$; is a labeled instruction of t and $inst$ is the instruction described at the beginning of the rule. The evaluation function $eval$ calculates the value of an arithmetic or boolean expression based on mem (ae stands for arithmetic expression). Sequence concatenation is denoted by $\circ$. The function $varsOfBuf$ takes a sequence of pairs and returns the set consisting of the first fields of these pairs.

$(\{com, rd\} \times \overrightarrow{x} \times \mathcal{D})$). Actions keep the information about the thread performing the transition and the actual parameters of the reads and the writes to shared variables. We are only interested in accesses to shared variables, therefore, other transitions are labelled with τ as thread local actions.

A TSO execution of a program $\mathcal{P}$ is a sequence of actions $\pi = \pi_1, \pi_2, \ldots, \pi_n$ such that there exists a sequence of states $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_n$, $\sigma_0 = s_0$ is the initial state of $\mathcal{P}$ and $\sigma_{i-1} \xrightarrow{\pi_i} \sigma_i$ is a valid transition for any $i \in \{1, \ldots, n\}$. We assume that buffers are empty at the end of the execution.

SC Semantics. Under SC, a program state is a pair of the form (pc, mem) where pc and mem are defined as above. Shared variables are read directly from the memory mem and every write updates directly the memory mem. To make the relationship between SC and TSO executions more obvious, every write instruction generates isu and com actions which follow one another in the

execution (each *isu* is immediately followed by the corresponding *com*). Since there are no write buffers, `fence` instructions have no effect under SC.

Traces and TSO Robustness. Consider a (TSO or SC) execution π of $\mathcal{P}$. The trace of π is a graph, denoted by $Tr(\pi)$: Nodes of $Tr(\pi)$ are actions of π except the τ actions. In addition, *isu* and *com* actions are unified in a single node. The *isu* action that puts an element into the buffer and the corresponding *com* action that drains that element from the buffer correspond to the same node in the trace. Edges of $Tr(\pi)$ represent the happens before order (hb) between these actions. The hb is union of four relations. The program order po keeps the order of actions performed by the same thread excluding the *com* actions. The store order so keeps the order of *com* actions on the same variable that write different values[1]. The read-from relation, denoted by rf, relates a *com* action to a rd action that reads its value. Lastly, the from-reads relation fr relates a rd action to a *com* action that overwrites the value read by rd; it is defined as the composition of rf and so.

We say that the program $\mathcal{P}$ is TSO robust if for any TSO execution π of $\mathcal{P}$, there exists an SC execution π' such that $Tr(\pi) = Tr(\pi')$. It has been proven that robustness implies that the program reaches the same valuations of the shared memory under both TSO and SC [7].

4 A Reduction Theory for Checking Robustness

We present a methodology for checking robustness which builds on concepts introduced in Lipton's reduction theory [18]. This theory allows to rewrite a given concurrent program (running under SC) into an equivalent one that has larger atomic blocks. Proving robustness is similar in spirit in the sense that one has to prove that issue and commit actions can happen together atomically. However, differently from the original theory, these actions do not correspond to different statements in the program (they are generated by the same write instruction). Nevertheless, we show that the concepts of left/right movers can be also used to prove robustness.

Movers. Let $\pi = \pi_1, \ldots, \pi_n$ be an SC execution. We say that the action π_i *moves right (resp., left)* in π if the sequence $\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \pi_i, \pi_{i+2}, \ldots, \pi_n$ (resp., $\pi_1, \ldots, \pi_{i-2}, \pi_i, \pi_{i-1}, \pi_{i+1} \ldots, \pi_n$) is also a valid execution of $\mathcal{P}$, the thread of π_i is different than the thread of π_{i+1} (resp., π_{i-1}), and both executions reach to the same end state σ_n. Since every issue action is followed immediately by the corresponding commit action, an issue action moves right, resp., left, when the commit action also moves right, resp., left, and vice-versa.

[1] Our definition of store order deviates slightly from the standard definition which relates any two writes writing on the same variable, independently of values. The notion of TSO trace robustness induced by this change is slightly weaker than the original definition, but still implies preservation of any safety property from the SC semantics to the TSO semantics. The results concerning TSO robustness used in this paper (Lemma 1) are also not affected by this change. See [8] for more details.

Let instOf_π be a function, depending on an execution π, which given an action $\pi_i \in \pi$, gives the labelled instruction that generated π_i. Then, a labelled instruction ℓ is a *right (resp., left) mover* if for all SC executions π of $\mathcal{P}$ and for all actions π_i of π such that $\mathsf{instOf}(\pi_i) = \ell$, π_i moves right (resp., left) in π.

A labelled instruction is a *non-mover* if it is neither left nor right mover, and it is a *both mover* if it is both left and right mover.

Reachability Between Instructions. An instruction ℓ' is *reachable from* the instruction ℓ if ℓ and ℓ' both belong to the same thread and there exists an SC execution π and indices $1 \leq i < j \leq |\pi|$ such that $\mathsf{instOf}_\pi(\pi_i) = \ell$ and $\mathsf{instOf}_\pi(\pi_j) = \ell'$. We say that ℓ' is reachable from ℓ *before a fence* if π_k is not an action generated by a `fence` instruction in the same thread as ℓ, for all $i < k < j$. When ℓ is a write instruction and ℓ' a read instruction, we say that ℓ' is *buffer-free* reachable from ℓ if π_k is not an action generated by a `fence` instruction in the same thread as ℓ or a write action on the same variable that ℓ' reads-from, for all $i < k < j$.

Definition 1. *We say that a write instruction ℓ_w is* atomic *if it is a left mover or every read instruction ℓ_r buffer-free reachable from ℓ_w is a right mover. We say that $\mathcal{P}$ is* write atomic *if every write instruction ℓ_w in $\mathcal{P}$ is atomic.*

Note that all of the notions used to define write atomicity (movers and instruction reachability) are based on SC executions of the programs. The following result shows that write atomicity implies robustness.

Theorem 1 (Soundness). *If $\mathcal{P}$ is write atomic, then it is robust.*

We will prove the contrapositive of the statement. For the proof, we need the notion of minimal violation defined in [7]. A minimal violation is a TSO execution in which the sum of the number of same thread actions between isu and corresponding com actions for all writes is minimal. A minimal violation is of the form $\pi = \pi_1, (t, isu), \pi_2, (t, rd, y, *), \pi_3, (t, com, x, *), \pi_4$ such that π_1 is an SC execution, only t can delay com actions, the first delayed action is the $(t, com, x, *)$ action after π_3 and it corresponds to (t, isu) after π_1, π_2 does not contain any com or $fence$ actions by t (writes of t are delayed until after $(t, rd, y, *)$), $(t, rd, y, *) \rightarrow_{hb^+} act$ for all $act \in \pi_3 \circ \{(t, com, x, *)\}$ (isu and com actions of other threads are counted as one action for this case), π_3 doesn't contain any action of t, π_4 contains only and all of the com actions of t that are delayed in $(t, isu) \circ \pi_2$ and no com action in $(t, com, x, *) \circ \pi_4$ touches y.

Minimal violations are important for us because of the following property:

Lemma 1 (Completeness of Minimal Violations [7]). *The program $\mathcal{P}$ is robust iff it does not have a minimal violation.*

Before going into the proof of Theorem 1, we define some notations. Let π be a sequence representing an execution or a fragment of it. Let Q be a set of thread identifiers. Then, $\pi|_Q$ is the projection of π on actions from the threads in Q. Similarly, $\pi|_n$ is the projection of π on first n elements for some number n. $sz(\pi)$ gives the length of the sequence π. We also define a product operator

$\otimes$. Let π and ρ be some execution fragments. Then, $\pi \otimes \rho$ is same as π except that if the i^{th} isu action of π is not immediately followed by a com action by the same thread, then i^{th} com action of ρ is inserted after this isu. The product operator helps us to fill unfinished writes in one execution fragment by inserting commit actions from another fragment immediately after the issue actions.

Proof (Theorem 1). Assume $\mathcal{P}$ is not robust. Then, there exists a minimal violation $\pi = \pi_1, \alpha, \pi_2, \theta, \pi_3, \beta, \pi_4$ satisfying the conditions described before, where $\alpha = (t, isu)$, $\theta = (t, rd, y, *)$ and $\beta = (t, com, x, *)$. Below, we show that the write instruction $w = \mathsf{instOf}(\alpha)$ is not atomic.

1. w is not a left mover.
   1.1. $\rho = \pi_1, \pi_2|_{\overrightarrow{t}\setminus\{t\}}, \pi_3|_{\overrightarrow{t}\setminus\{t\}}|_{sz(\pi_3|_{\overrightarrow{t}\setminus\{t\}})-1}, \gamma, (\alpha, \beta)$ is an SC execution of $\mathcal{P}$ where γ is the last action of π_3. γ is a read or write action on x performed by a thread t' other than t and value of γ is different from what is written by β.
      1.1.1. ρ is an SC execution because t never changes value of a shared variable in π_2 and π_3. So, even we remove actions of t in those parts, actions of other threads are still enabled. Since other threads perform only SC operations in π, $\pi_1, \pi_2|_{\overrightarrow{t}\setminus\{t\}}, \pi_3|_{\overrightarrow{t}\setminus\{t\}}$ is an SC execution. From π, we also know that the first enabled action of t is α if we delay the actions of t in π_2 and π_3.
      1.1.2. The last action of π_3 is γ. By definition of a minimal violation, we know that $\theta \rightarrow_{hb^+} \alpha$ and π_3 does not contain any action of t. So, there must exist an action $\gamma \in \pi_3$ such that either γ reads from x and $\gamma \rightarrow_{fr} \beta$ in π or γ writes to x and $\gamma \rightarrow_{st} \beta$ in π. Moreover, γ is the last action of π_3 because if there are other actions after γ, we can delete them and can obtain another minimal violation which is shorter than π and hence contradict the minimality of π.
   1.2. $\rho' = \pi_1, \pi_2|_{\overrightarrow{t}\setminus\{t\}}, \pi_3|_{\overrightarrow{t}\setminus\{t\}}|_{sz(\pi_3|_{\overrightarrow{t}\setminus\{t\}})-1}, (\alpha, \beta), \gamma$ is an SC execution with a different end state than ρ defined in 1.1 has or it is not an SC execution, where $\mathsf{instOf}(\gamma') = \mathsf{instOf}(\gamma)$.
      1.2.1. In the last state of ρ, x has the value written by β. If γ is a write action on x, then x has a different value at the end of ρ' due to the definition of a minimal violation. If γ is a read action on x, then it does not read the value written by β in ρ. However, γ reads this value in ρ' . Hence, ρ' is not a valid SC execution.
2. There exists a read instruction r buffer-free reachable from w such that r is not a right mover. We will consider two cases: Either there exists a rd action of t on variable z in π_2 such that there is a later write action by another thread t' on z in π_2 that writes a different value or not. Moreover, z is not a variable that is touched by the delayed commits in π_4 i.e., it does not read its value from the buffer.
   2.1. We first evaluate the negation of above condition. Assume that for all actions γ and γ' such that γ occurs before γ' in π_2, either $\gamma \neq (t, rd, z, v_z)$ or $\gamma' \neq (t', isu)(t', com, z, v'_z)$. Then, $r = \mathsf{instOf}(\theta)$ is not a right mover and it is buffer-free reachable from w.

2.1.1. $\rho = \pi_1, \pi_2|_{\overrightarrow{t}\setminus\{t\}}, \pi_2|_{\{t\}} \otimes \pi_4, \theta, \theta'$ is a valid SC execution of $\mathcal{P}$ where $\theta' = (t', isu)(t', com, y, *)$ for some $t \neq t'$.

2.1.1.1. ρ is an SC execution. $\pi_1, \pi_2|_{\overrightarrow{t}\setminus\{t\}}$ is a valid SC execution since t does not update value of a shared variable in π_2. Moreover, all of the actions of t become enabled after this sequence since t never reads value of a variable updated by another thread in π_2. Lastly, the first action of π_3 is enabled after this sequence.

2.1.1.2. The first action of π_3 is $\theta' = (t', isu)(t', com, y, *)$. Let θ' be the first action of π_3. Since $\theta \rightarrow_{hb} \theta'$ in π and θ' is not an action of t by definition of minimal violation, the only case we have is $\theta \rightarrow_{fr} \theta'$. Hence, θ' is a write action on y that writes a different value than θ reads.

2.1.1.3. r is buffer-free reachable from w. ρ is a SC execution, first action of ρ after $\pi_1, \pi_2|_{\overrightarrow{t}\setminus\{t\}}$ is α, β; $w = \mathsf{instOf}((\alpha, \beta))$, $r = \mathsf{instOf}(\theta)$ and actions of t in ρ between α, β and θ are not instances of a fence instruction or write to y.

2.1.2. $\rho' = \pi_1, \pi_2|_{\overrightarrow{t}\setminus\{t\}}, \pi_2|_{\{t\}} \otimes \pi_4, \theta', \theta$ is not a valid SC execution.

2.1.2.1. In the last state of ρ, the value of y seen by t is the value read in θ. It is different than the value written by θ'. However, at the last state of ρ', the value of y t sees must be the value θ' writes. Hence, ρ' is not a valid SC execution.

2.2. Assume that there exists $\gamma = (t, rd, z, v_z)$ and $\gamma' = (t', isu)(t', com, z, v'_z)$ in π_2. Then, $r = \mathsf{instOf}(\gamma)$ is not a right mover and r is buffer-free reachable from w.

2.2.1. Let i be the index of γ and j be the index of γ' in π_2. Then, define $\rho = \pi_1, \pi_2|_{j-1}|_{\overrightarrow{t}\setminus\{t\}}, \pi_2|_i|_{\{t\}} \otimes \pi_4, \gamma'$. ρ is an SC execution of $\mathcal{P}$.

2.2.1.1. ρ is an SC execution. $\pi_1, \pi_2|_{j-1}|_{\overrightarrow{t}\setminus\{t\}}$ prefix is a valid SC execution because t does not update any shared variable in π_2. Moreover, all of the actions of t in $\pi_2|_i|_{\{t\}} \otimes \pi_4$ become enabled after this sequence since t never reads a value of a variable updated by another thread in π_2 and γ' is the next enabled in π_2 after this sequence since it is a write action.

2.2.2. Let i and j be indices of γ and γ' in π_2 respectively. Define $\rho' = \pi_1, \pi_2|_{j-1}|_{\overrightarrow{t}\setminus\{t\}}, \pi_2|_{i-1}|_{\{t\}} \otimes \pi_4, \gamma', \gamma$. Then, ρ' is not a valid SC execution.

2.2.2.1. In the last state of ρ, value of z seen by t is v_z. It is different than the v'_z, value written by γ'. However, in the last state of ρ', the value of z t sees must be v'_z. Hence, ρ' is not a valid SC execution.

2.2.3. r is buffer-free reachable from w because ρ defined in 2.2.1 is an SC execution, first action after $\pi_1, \pi_2|_{j-1}|_{\overrightarrow{t}\setminus\{t\}}$ is α, β, $w = \mathsf{instOf}((\alpha, \beta))$, $r = \mathsf{instOf}(\gamma)$ and actions of t in ρ between α, β and θ are not instances of a fence instruction or a write to z by t.

5 Abstractions and Verifying non-Robust Programs

In this section, we introduce program abstractions which are useful for verifying non-robust TSO programs (or even robust programs – see an example at the end of this section). In general, a program $\mathcal{P}'$ abstracts another program $\mathcal{P}$ for some semantic model $\mathbb{M} \in \{\mathrm{SC}, \mathrm{TSO}\}$ if every shared variable valuation σ reachable from the initial state in an $\mathbb{M}$ execution of $\mathcal{P}$ is also reachable in an $\mathbb{M}$ execution of $\mathcal{P}'$. We denote this abstraction relation as $\mathcal{P} \preceq_{\mathbb{M}} \mathcal{P}'$.

In particular, we are interested in *read instruction abstractions*, which replace instructions that read from a shared variable with more "liberal" read instructions that can read more values (this way, the program may reach more shared variable valuations). We extend the program syntax in Section 3 with havoc instructions of the form $\texttt{havoc}(\langle reg \rangle, \langle varbexpr \rangle)$, where $\langle varbexpr \rangle$ is a boolean expression over a set of registers and a single shared variable $\langle var \rangle$. The meaning of this instruction is that the register reg is assigned with any value that satisfies $varbexpr$ (where the other registers and the variable var are interpreted with their current values). The program abstraction we consider will replace read instructions of the form $\langle reg \rangle := \langle var \rangle$ with havoc instructions $\texttt{havoc}(\langle reg \rangle, \langle varbexpr \rangle)$.

While replacing read instructions with havoc instructions, we must guarantee that the new program reaches at least the same set of shared variable valuations after executing the havoc as the original program after the read. Hence, we allow such a rewriting only when the boolean expression $varbexpr$ is weaker (in a logical sense) than the equality $reg = var$ (hence, there exists an execution of the havoc instruction where $reg = var$).

Lemma 2. *Let $\mathcal{P}$ be a program and $\mathcal{P}'$ be obtained from $\mathcal{P}$ by replacing an instruction $l_1 : x := r; \texttt{goto}\ l_2$ of a thread t with $l_1 : \texttt{havoc}(r, \phi(x, \overrightarrow{r})); \texttt{goto}\ l_2$ such that $\forall x, r.\ x = r \implies \phi(x, \overrightarrow{r})$ is valid. Then, $\mathcal{P} \preceq_{SC} \mathcal{P}'$ and $\mathcal{P} \preceq_{TSO} \mathcal{P}'$.*

The notion of trace extends to programs that contain havoc instructions as follows. Assume that $(t, hvc, x, \phi(x))$ is the action generated by an instruction $\texttt{havoc}(r, \phi(x, \overrightarrow{r}))$, where x is a shared variable and $\overrightarrow{r}$ a set of registers (the action stores the constraint ϕ where the values of the registers are instantiated with their current values – the shared variable x is the only free variable in $\phi(x)$). Roughly, the hvc actions are special cases of rd actions. Consider an execution π where an action $\alpha = (t, hvc, x, \phi(x))$ is generated by reading the value of a write action $\beta = (com, x, v)$ (i.e., the value v was the current value of x when the havoc instruction was executed). Then, the trace of π contains a read-from edge $\beta \rightarrow_{rf} \alpha$ as for regular read actions. However, fr edges are created differently. If α was a rd action we would say that we have $\alpha \rightarrow_{fr} \gamma$ if $\beta \rightarrow_{rf} \alpha$ and $\beta \rightarrow_{st} \gamma$. For the havoc case, the situation is a little bit different. Let $\gamma = (com, x, v')$ be an action. We have $\alpha \rightarrow_{fr} \gamma$ if and only if either $\beta \rightarrow_{rf} \alpha$, $\beta \rightarrow_{st} \gamma$ and $\phi(v')$ is false or $\alpha \rightarrow_{fr} \gamma'$ and $\gamma' \rightarrow_{st} \gamma$ where γ' is an action. Intuitively, there is a from-read dependency from an havoc action to a commit action, only when the commit action invalidates the constraint $\phi(x)$ of the havoc (or if it follows such a commit in store order).

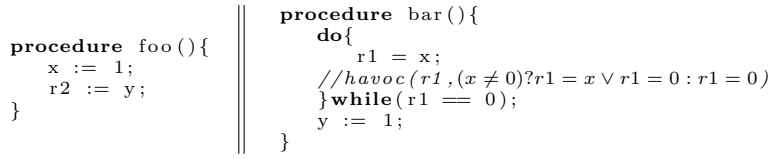

Fig. 6. An example program that needs a read abstraction to pass our robustness checks. The `havoc` statement in comments reads as follows: if value of x is not 0 then $r1$ gets either the value of x or 0. Otherwise, it is 0.

The notion of write-atomicity (Definition 1) extends to programs with havoc instructions by interpreting havoc instructions $\texttt{havoc}(r, \phi(x, \overrightarrow{r}))$ as regular read instructions $r := x$. Theorem 1 which states that write-atomicity implies robustness can also be easily extended to this case.

Read abstractions are useful in two ways. First, they allow us to prove properties of non-robust program as the work stealing queue example in Figure 3. We can apply appropriate read abstractions to relax the original program so that it becomes robust in the end. Then, we can use SC reasoning tools on the robust program to prove invariants of the program.

Second, read abstractions could be helpful for proving robustness directly. The method based on write-atomicity we propose for verifying robustness is sound but not complete. Some incompleteness scenarios can be avoided using read abstractions. If we can abstract read instructions such that the new program reaches exactly the same states (in terms of shared variables) as the original one, it may help to avoid executions that violate mover checks.

Consider the program in Figure 6. The write statement `x := 1` in procedure `foo` is not atomic. It is not a left mover due to the read of x in the do-while loop of `bar`. Moreover, the later read from y is buffer-free reachable from this write and it is not a right mover because of the write to y in `bar`. To make it atomic, we apply read abstraction to the read instruction of `bar` that reads from x. In the new relaxed read, $r1$ can read 0 along with the value of x when x is not zero as shown in the comments below the instruction. With this abstraction, the write to x becomes a left mover because reads from x after the write can now read the old value which was 0. Thus, the program becomes write-atomic. If we think of TSO traces of the abstract program and replace *hvc* nodes with *rd* nodes, we get exactly the TSO traces of the original program. However, the abstraction adds more SC traces to the program and the program becomes robust.

6 Experimental Evaluation

To test the practical value of our method, we have considered the benchmark for checking TSO robustness described in [2], which consists of 34 programs. This benchmark is quite exhaustive, it includes examples introduced in previous works on this subject. Many of the programs in this benchmark are easy to prove being write-atomic. Every write is followed by no buffer-free read instruction which makes them trivially atomic (like the message passing program in Figure 1). This holds for 20 out of the 34 programs. Out of the remaining programs, 13 required mover checks and 4 required read abstractions to show robustness

Table 1. Benchmark results. The second column (RB) stands for the robustness status of the original program according to our extended *hb* definition. RA column shows the number of read abstractions performed. RM column represents the number of read instructions that are checked to be right movers and the LM column represents the write instructions that are shown to be left movers. PO shows the total number of proof obligations generated and VT stands for the total verification time in seconds.

Name	RB	RA	RM	LM	PO	VT
Chase-Lev:	-	1	2	-	149	0.332
FIFO-iWSQ:	+	-	2	-	124	0.323
LIFO-iWSQ:	+	-	1	-	109	0.305
Anchor- iWSQ:	+	-	1	-	109	0.309
MCSLock:	+	2	2	-	233	0.499
r+detour:	+	-	1	-	53	0.266
r+detours:	+	-	1	-	64	0.273
sb+detours+coh:	+	-	2	-	108	0.322
sb+detours:	+	-	1	1	125	0.316
write+r+coh:	+	-	1	-	78	0.289
write+r:	+	-	1	-	48	0.261
dc-locking:	+	1	4	1	52	0.284
inline_pgsql:	+	2	2	-	90	0.286

(our method didn't succeed on one of the programs in the benchmark, explained at the end of this section). Except Chase-Lev, the initial versions of all the 12 examples are trace robust[2]. Besides Chase-Lev, read-abstractions are equivalent to the original programs in terms of reachable shared variable configurations. Detailed information for these examples can be found in Table 1.

To check whether writes/reads are left/right movers and the soundness of abstractions, we have used the tool Civl [13]. This tool allows to prove assertions about concurrent programs (Owicki-Gries annotations) and also to check whether an instruction is a left/right mover. The buffer-free read instructions reachable from a write before a fence were obtained using a trivial analysis of the control-flow graph (CFG) of the program. This method is a sound approximation of the definition in Section 4 but it was sufficient for all the examples.

Our method was not precise enough to prove robustness for only one example, named as `nbw-w-lr-rl` in [7]. This program contains a method with explicit calls to the `lock` and `unlock` methods of a spinlock. The instruction that writes to the lock variable inside the `unlock` method is not atomic, because of the reads from the lock variable and the calls to the `getAndSet` primitive inside the `lock` method. Abstracting the reads from the lock variable is not sufficient in this case due to the conflicts with `getAndSet` actions. However, we believe that read abstractions could be extended to `getAndSet` instructions (which both read and write to a shared variable atomically) in order to deal with this example.

[2] If we consider the standard notion of *so* (that relates any two writes on the same variable independent of their values), all examples except `MCSLock` and `dc-locking` become non trace robust.

7 Related Work

The weakest correctness criterion that enables SC reasoning for proving invariants of programs running under TSO is *state-robustness* i.e., the reachable set of states is the same under both SC and TSO. However, this problem has high complexity (non-primitive recursive for programs with a finite number of threads and a finite data domain [6]). Therefore, it is difficult to come up with an efficient and precise solution. A symbolic decision procedure is presented in [1] and over-approximate analyses are proposed in [14, 15].

Due to the high complexity of state-robustness, stronger correctness criteria with lower complexity have been proposed. Trace-robustness (that we call simply robustness in our paper) is one of the most studied criteria in the literature. Bouajjani et al. [9] have proved that deciding trace-robustness is PSpace-complete, resp., EXPSpace-complete, for a finite, resp., unbounded, number of threads and a finite data domain.

There are various tools for checking trace-robustness. Trencher [7] applies to bounded-thread programs with finite data. In theory, the approach in Trencher can be applied to infinite-state programs, but implementing it is not obvious because it requires solving non-trivial reachability queries in such programs. In comparison, our approach (and our implementation based on Civl) applies to infinite state programs. All our examples consider infinite data domains, while Chase-Lev, FIFO-iWSQ, LIFO-iWSQ, Anchor-iWSQ, MCSLock, dc-locking and inline_pgsql have an unbounded number of threads. Musketeer [4] provides an approximate solution by checking existence of critical cycles on the control-flow graph. While Musketeer can deal with infinite data (since data is abstracted away), it is restricted to bounded-thread programs. Thus, it cannot deal with the unbounded thread examples mentioned above. Furthermore, Musketeer cannot prove robust even some examples with finitely many threads, e.g., nbw_w_wr, write+r, r+detours, sb+detours+coh. Other tools for approximate robustness checking, to which we compare in similar ways, have been proposed in [5, 10, 11].

Besides trace-robustness, there are other correctness criteria like triangular race freedom (Trf) and persistence that are stronger than state-robustness. Persistence ([2]) is incomparable to trace-robustness, and Trf [19] is stronger than both trace-robustness and persistence. Our method can verify examples that are state-robust but neither persistent nor Trf.

Reduction and abstraction techniques were used for reasoning on SC programs. Qed ([12]) is a tool that supports statement transformations as a way of abstracting programs combined with a mover analysis. Also, Civl ([13]) allows proving location assertions in the context of the Owicki-Gries logic which is enhanced with Lipton's reduction theory [17]. Our work enables the use of such tools for reasoning about the TSO semantics of a program.

References

1. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under tso. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 204–219. Springer, 2012.
2. Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. The best of both worlds: Trading efficiency and optimality in fence insertion for tso. In *European Symposium on Programming Languages and Systems*, pages 308–332. Springer, 2015.
3. Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, 1993.
4. Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Dont sit on the fence. In *International Conference on Computer Aided Verification*, pages 508–524. Springer, 2014.
5. Jade Alglave and Luc Maranget. Stability in weak memory models. In *Computer Aided Verification*, pages 50–66. Springer, 2011.
6. Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. *ACM Sigplan Notices*, 45(1):7–18, 2010.
7. Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against tso. In *Programming Languages and Systems*, pages 533–553. Springer, 2013.
8. Ahmed Bouajjani, Constantin Enea, Suha Orhun Mutluergil, and Serdar Tasiran. Reasoning about tso programs using reduction and abstraction. *CoRR*, abs/1804.05196, 2018.
9. Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In *International Colloquium on Automata, Languages, and Programming*, pages 428–440. Springer, 2011.
10. Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *International Conference on Computer Aided Verification*, pages 107–120. Springer, 2008.
11. Jabob Burnim, Koushik Sen, and Christos Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 11–25. Springer, 2011.
12. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *ACM Symposium on Principles of Programming Languages*, page 14. Association for Computing Machinery, Inc., January 2009.
13. Chris Hawblitzel, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. *Computer Aided Verification*, 2015.
14. Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In *ACM SIGPLAN Notices*, volume 46, pages 187–198. ACM, 2011.
15. Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *ACM SIGACT News*, 43(2):108–123, 2012.
16. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.
17. Richard J Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

18. Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
19. Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *ECOOP*, volume 6183, pages 478–503. Springer, 2010.
20. Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.
21. Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):282–312, 1988.