Checking and Enforcing Robustness against TSO

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

[1] LIAFA, University Paris 7
[2] Fraunhofer ITWM
[3] University of Kaiserslautern
abou@liafa.univ-paris-diderot.fr,
{derevenetc,meyer}@cs.uni-kl.de

Abstract. We present algorithms for checking and enforcing robustness of concurrent programs against the Total Store Ordering (TSO) memory model. A program is robust if all its TSO computations correspond to computations under the Sequential Consistency (SC) semantics.

We provide a complete characterization of non-robustness in terms of so-called attacks: a restricted form of (harmful) out-of-program-order executions. Then, we show that detecting attacks can be parallelized, and can be solved using state reachability queries under the SC semantics in a suitably instrumented program obtained by a linear size source-to-source translation. Importantly, the construction is valid for an unbounded number of memory addresses and an arbitrary number of parallel threads. It is independent from the data domain and from the size of store buffers in the TSO semantics. In particular, when the data domain is finite and the number of addresses is fixed, we obtain decidability and complexity results for robustness, even for a parametric number of threads.

As a second contribution, we provide an algorithm for computing an optimal set of fences that enforce robustness. We consider two criteria of optimality: minimization of program size and maximization of its performance. The algorithms we define are implemented, and we successfully applied them to analyzing and correcting several concurrent algorithms.

1 Introduction

Sequential Consistency (SC) [21] is a natural shared-memory model where the actions of different threads are interleaved while the program order between actions of each thread is preserved. For performance reasons, however, modern multiprocessors implement weaker memory models relaxing program order. For instance, the common store-to-load relaxation, which allows loads to overtake earlier stores, reflects the use of *store buffers*. It is actually the main feature of the TSO (Total Store Ordering) model adopted, e.g., in x86 machines [28].

Nonetheless, programmers often assume that memory accesses are performed according to SC, where they are instantaneous and atomic. This assumption is safe for *data-race-free* programs [3], but in many situations data-race-freedom does not apply. This is, for instance, the case of programs implementing synchronization operations, concurrency libraries, and other performance-critical system services employing lock-free synchronization. These programs are designed to be

M. Felleisen and P. Gardner (Eds.): ESOP 2013, LNCS 7792, pp. 533–553, 2013.
© Springer-Verlag Berlin Heidelberg 2013

robust against relaxations, i.e., relaxations should not introduce behaviors that are impossible under SC. Memory fences must be included appropriately in programs to prevent non-SC behaviors. Getting such programs right is a notoriously difficult and error-prone task. Therefore, important issues in this context are (1) checking robustness of a program against the relaxations of a memory model, and (2) identifying a set of program locations where it is *necessary* to insert fences to ensure robustness.

In this paper we address these two issues in the case of TSO. We consider a general setting without any bounds on the shared-memory size, nor on the size of the store buffers in the TSO semantics, nor on the number of threads. This allows us to reason about robustness of general algorithms without assuming any fixed values for these parameters that depend on the actual machine's implementation. Moreover, we tackle these issues for general programs, independently from the domain of data they manipulate.

Robustness against memory models has been addressed first by Burckhardt and Musuvathi in [10] (actually, for TSO only), and subsequently by Burnim et al. in [11]. Alglave and Maranget developed a general framework for reasoning about robustness against memory models in [4,5] (where the term *stability* is used instead of robustness). Roughly, these works are based on characterizing robustness in terms of acyclicity of a suitable happens-before relation. In that, they follow Shasha and Snir [29] who introduced a notion of *trace* that captures the control and data dependencies between events of an SC computation, and established that computations that are not SC have a cyclic happens-before relation. We adopt here the same notion of (trace-)robustness, i.e., a program is robust if every TSO computation has the same trace as some SC computation.

From an algorithmic point of view, the existing works mentioned above *do not provide decision procedures* for robustness. [10,11] provide testing procedures based on enumerating TSO runs and checking that they do not produce happens-before cycles. Clearly, while these procedures can establish non-robustness, they can never prove a program robust. On the other hand, [5] provides a sound over-approximate static analysis that allows for proving robustness, but may also inaccurately conclude to non-robustness and insert fences unnecessarily. We are interested here in developing an approach that allows for precise checking of trace-robustness, and for optimal fence insertion (in a sense defined later).

In our previous work [9], trace-robustness against TSO has been proven to be decidable and PSPACE-complete, even for unbounded store buffers, in the case of a fixed number of threads and assuming a fixed number of shared variables, ranging over a finite data domain. The method that shows this decidability and complexity result does not provide a practical algorithm: it is based on a non-deterministic, bounded enumeration of computations. Moreover, it does not carry over to the general setting we consider here. Therefore, in this paper we propose a novel approach to checking robustness that is fundamentally different from [9]. We provide a general, source-to-source reduction of the trace-robustness problem against TSO to the *state reachability problem under the SC semantics*. In other words, we show that trace-robustness is not more expensive than SC

state reachability, which is the unavoidable problem to be solved by any precise decision algorithm for concurrent programs. This is the *key contribution of the paper* from which we derive other results, such as decidability results in particular cases, as well as an algorithm for efficient fence insertion.

To establish our reduction, we first provide a complete characterization of non-robustness in terms of so-called *feasible attacks*. An attack is a pair of load and store instructions of a thread, called the attacker, whose reordering may lead to a non-SC computation. In that case we say the attack is feasible, because it has a (TSO) witness computation. The special form of witness computations allows us to detect them by solving an SC state reachability query in an *instrumented program*. The fact that *only the SC semantics* (of the instrumented program) needs to be considered for detecting non-SC behaviors (of the original program) is important: it relieves us of examining TSO computations, which obliges one to encode (somehow) the contents of store buffers (as in, e.g., [10,11]). Interestingly, the feasibility checks for different attacks can be parallelized, which speeds up the decision procedure. Overall, we provide a reduction of the TSO robustness problem to a quadratic number (in the size of the program) of state reachability queries under the SC semantics in linear-size instrumented programs of the same type as the original one. Our construction is source-to-source and is valid for (1) an unbounded number of memory addresses/variables, (2) an arbitrary data domain, (3) an arbitrary number of threads, and (4) unbounded store buffers.

With this reduction, we can harness all techniques and tools that are available for solving SC reachability queries (either exactly, or approximately) in various classes of concurrent programs, regardless of decidability and complexity issues. It also yields decision algorithms for significant classes of programs. Assume we have a finite number of memory addresses, and the data domain is finite. Then, for a fixed number of threads, a direct consequence of our reduction is that the robustness problem is decidable and in PSpace since it is polynomially reducible to state reachability in finite-state concurrent programs. Therefore, we obtain in this case a *simple* robustness checking algorithm which matches the complexity upper bound proven in [9]. Our construction also provides an effective decision algorithm for the up to now open case where the *number of threads is parametric*. In this case, SC state reachability queries can be solved in vector addition systems with states (VASS), or equivalently as coverability in Petri nets, which is known to be decidable [27] and EXPSpace-hard [24]. In both cases (fixed and parametric number of threads) the decision algorithms do not assume bounded store buffers.

As last contribution, we address the issue of enforcing robustness by fence insertion. Obviously, inserting a fence after each store ensures robustness, but it also ruins all performance benefits that a relaxed memory model brings. A natural requirement on the set of fences is irreducibility, i.e., minimality wrt. set inclusion. Since there may be several irreducible sets enforcing robustness, it is natural to ask for a set that optimizes some notion of cost. We assume that we have a *cost function* that defines the cost of inserting a fence at each program location. For instance, by assuming cost 1 for all locations, we optimize the size

of the fence set. Other cost functions reflect the performance of the resulting program. We propose an algorithm which, given a cost function, computes an optimal set of fences. The algorithm is based on 0/1-integer linear programming and exploits the notion of attacks to guide the selection of fences.

We implemented the algorithms (using SPIN as a back-end SC reachability checker), and applied them successfully to checking and correcting a number of examples, including mutual exclusion protocols and concurrent data structures. The experiments we have carried out show that our approach is quite effective: (1) many of the attacks to be examined can be discarded by simple syntactic checks (e.g., the presence of a fence between the store and load instructions), and those that require solving reachability queries are handled in few seconds, (2) the fence insertion procedure finds efficiently optimal sets of fences, in particular, it can handle the version of the Non-Blocking Write protocol [18] described in [25] (where the write is guarded by a Linux x86 spinlock) for which Owens' method based on so-called *triangular data races* (see below) inserts unnecessary fences.

Related Work. There are only few results on decidability and complexity of relaxed memory models. Reachability under TSO has been shown to be decidable but non-primitive recursive [7] in the case of a finite number of threads and a finite data domain. In the same setting, trace-robustness has been shown to be PSPACE-complete [9] using a combinatorial approach.

Alur et al. have shown that checking sequential consistency of a concurrent implementation wrt. a specification is undecidable in general [6]. This result does not contradict our findings: we consider here the special case where the implementation is the TSO semantics and the specification is the SC semantics of the same program. In [15], the problem of deciding whether a given computation is SC feasible has been proven NP-complete. Robustness is concerned with all TSO computations, instead.

As mentioned above, the problem of checking and enforcing trace-robustness against weak memory models has been addressed in [10,11,5], but none of these works provide (sound and complete) decision procedures. Owens proposes in [25] a notion of robustness that is *stronger* than trace-robustness, based on detecting triangular data races. This allows for sound trace-robustness checking but, as Owens shows in his paper, in some cases leads to unnecessary fences which can be harmful for performance. Moreover, the notion of triangular data races comes without an algorithm for checking it[1]. Complexity considerations (using the techniques in [9]) show that detecting triangular data races requires solving state reachability queries under SC. Therefore, as we show here, checking trace-robustness is not more expensive than detecting triangular data races.

State-based robustness (which preserves the reachable states) is a weaker robustness criterion, but does not preserve path properties in contrast to trace-robustness, and is of significantly higher complexity (non-primitive recursive as it can be deduced from [7], instead of PSPACE). It has been addressed in a

[1] Citation from [25]: "*... formal reasoning directly on traces can be tedious, so a program logic or sound static analyzer specialized to proving triangular-race freedom might make the application of TRF more convenient.*"

precise manner in [2,23], where symbolic decision procedures together with fence insertion algorithms are provided. The same issue is addressed in [19,20] using over-approximate reachability analysis based on abstractions of the store buffers.

Finally, let us mention work that considers the dual approach: starting from a robust program, remove unnecessary fences [30]. This work is aimed at compiler optimizations and does not provide a decision procedure for robustness. It can also not find an optimal set of fences that enforce trace-robustness.

2 Parallel Programs

Syntax. We consider parallel programs with shared memory as defined by the grammar in Figure 1. Programs have a name and consist of a finite number of threads. Each thread has an identifier and a list of local registers it operates on. The thread's source code is given as a finite sequence of labelled instructions. More than one instruction can be marked by the same label; this allows us to implement conditional branching, iteration, and non-determinism in a lightweight syntax. The instruction set includes loads from memory to a local register, stores to memory, memory fences to control the TSO store buffers, local computations, and assertions. Figure 2 provides a sample program.

$\langle prog \rangle$ ::= `program` $\langle pid \rangle$ $\langle thrd \rangle^*$
$\langle thrd \rangle$ ::= `thread` $\langle tid \rangle$
`regs` $\langle reg \rangle^*$
`init` $\langle label \rangle$
`begin` $\langle linst \rangle^*$ `end`
$\langle linst \rangle$ ::= $\langle label \rangle$`:` $\langle inst \rangle$`; goto` $\langle label \rangle$`;`
$\langle inst \rangle$ ::= $\langle reg \rangle \leftarrow$ `mem[`$\langle expr \rangle$`]`
| `mem[`$\langle expr \rangle$`]` $\leftarrow \langle expr \rangle$
| `mfence`
| $\langle reg \rangle \leftarrow \langle expr \rangle$
| `assert` $\langle expr \rangle$
$\langle expr \rangle$ ::= $\langle fun \rangle$`(`$\langle reg \rangle^*$`)`

Fig. 1. Syntax of parallel programs

```
program Dekker
thread t1 regs r1 init l0 begin
l0: mem[x] ← 1; goto l1;
l1: r1 ← mem[y]; goto l2;
end
thread t2 regs r2 init l'0 begin
l'0: mem[y] ← 1; goto l'1;
l'1: r2 ← mem[x]; goto l'2;
end
```

Fig. 2. Simplified version of Dekker's mutex algorithm. Under SC, it is impossible that $r_1 = r_2 = 0$ when both threads reach l_2 and l'_2.

We assume a program comes with two sets: a *data domain* DOM and a *function domain* FUN. The data domain should contain value zero: $0 \in \mathsf{DOM}$. Moreover, we assume that all values from DOM can be used as addresses. Each memory location accessed by loads and stores is identified by such an address, and memory locations identified by different addresses do not overlap. The set FUN contains functions that are defined on the data domain and can be used in the program. Note that we do not make any finiteness assumptions.

TSO Semantics. Fix a program $\mathcal{P}$ with threads $\mathsf{THRD} := \{t_1, \ldots, t_n\}$. Let each thread t_i have the initial label $\mathsf{l}_{0,i}$ and declare registers $\overline{r_i}$. We define the set of variables as the union of addresses and registers: $\mathsf{VAR} := \mathsf{DOM} \cup \cup_{i \in [1,n]} \overline{r_i}$. We denote the set of all instruction labels that occur in threads by LAB.

The TSO semantics we define is operational, in terms of labelled transitions between states. On the x86 TSO architecture, each thread effectively has a local FIFO buffer that keeps stores for later execution [26,28,10,11]. Therefore, a *state* is a triple $\mathsf{s} = (\mathsf{pc}, \mathsf{val}, \mathsf{buf})$ where the program counter $\mathsf{pc}\colon \mathsf{THRD} \to \mathsf{LAB}$ holds, for each thread, the label of the instruction(s) to be executed next. The valuation $\mathsf{val} : \mathsf{VAR} \to \mathsf{DOM}$ gives the values of registers and memory locations. The third component $\mathsf{buf}\colon \mathsf{THRD} \to (\mathsf{DOM} \times \mathsf{DOM})^*$ is the (per thread) buffer content: a sequence of address-value pairs $a \leftarrow v$.

In the *initial state* $\mathsf{s}_0 := (\mathsf{pc}_0, \mathsf{val}_0, \mathsf{buf}_0)$ the program counter is set to the initial labels, $\mathsf{pc}_0(t_i) := \mathsf{l}_{0,i}$ for all $t_i \in \mathsf{THRD}$, registers and addresses hold value zero, $\mathsf{val}_0(x) := 0$ for all $x \in \mathsf{VAR}$, and all buffers are empty, $\mathsf{buf}_0(t) := \varepsilon$ for all $t \in \mathsf{THRD}$. Here, ε denotes the empty sequence.

Instructions yield transitions between states that are labelled by *actions* from $\mathsf{ACT} := \mathsf{THRD} \times (\{\mathsf{isu}, \mathsf{loc}\} \cup (\{\mathsf{ld}, \mathsf{st}\} \times \mathsf{DOM} \times \mathsf{DOM}))$. An action consists of the thread id and the actual arguments of the executed instruction. We use loc to abstract assignments and asserts that are local to a thread. An issue action isu indicates that a store was executed by a thread. The store action (t, st, a, v) gives the moment when the store becomes visible in memory.

The *TSO transition relation* $\rightarrow_{\mathsf{TSO}}$ is the smallest relation between TSO states that is defined by the rules in Table 1. The rules repeat, up to notation and support for locked instructions, Figure 1 from [26]. The first two rules implement loads from the buffer and from the memory, respectively. By the third rule, store instructions enqueue write operations to the buffer. The fourth rule non-deterministically dequeues and executes them on memory. The fifth rule defines that memory fences can only be executed when the buffer is empty. The last two rules refer to local assignments and assertions. We omit locked instructions to keep things simple. Introducing them is straightforward and does not affect the results. Indeed, our implementation supports them [1].

The set of *TSO computations* contains all sequences of actions that lead from the initial TSO state to a state where all buffers are empty:

$$\mathsf{C}_{\mathsf{TSO}}(\mathcal{P}) := \{\tau \in \mathsf{ACT}^* \mid \mathsf{s}_0 \xrightarrow{\tau}_{\mathsf{TSO}} \mathsf{s} \text{ for some TSO state } \mathsf{s} = (\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \text{ with } \mathsf{buf}(t) = \varepsilon \text{ for all } t \in \mathsf{THRD}\}.$$

The requirement of empty buffers is not important for our results but rather a modelling choice. Figure 3 presents a TSO computation of Dekker's program where the store of the first thread is delayed past the load.

$$\tau = \overbrace{(t_1, \mathsf{isu})\,(t_1, \mathsf{ld}, \mathsf{y}, 0)\,(t_2, \mathsf{isu})\,(t_2, \mathsf{st}, \mathsf{y}, 1)\,(t_2, \mathsf{ld}, \mathsf{x}, 0)\,(t_1, \mathsf{st}, \mathsf{x}, 1)}$$

Fig. 3. A TSO computation of Dekker's algorithm. The arc connects the issue action with the corresponding delayed store action of the first thread.

SC Semantics. Under SC [21], stores are not buffered and hence states are pairs $(\mathsf{pc}, \mathsf{val})$. The rules for SC transitions are appropriately simplified TSO rules. To avoid case distinctions between TSO and SC in the definition of traces, a store

Table 1. TSO transition rules, assuming $\mathsf{pc}(t) = \mathsf{l}$, an instruction $\langle instr \rangle$ at label l with destination l', and $\mathsf{pc}' := \mathsf{pc}[t := \mathsf{l}']$. We use $\downarrow$ to denote projection and $*$ for any value, i.e., $\mathsf{buf}(t)\downarrow(a \leftarrow *)$ is the list of address-value pairs in the buffer of thread t that have address a.

$$\frac{\langle instr \rangle = r \leftarrow \texttt{mem}[f_a(\overline{r_a})],\ a = f_a(\mathsf{val}(\overline{r_a})),\ \mathsf{buf}(t)\downarrow(a \leftarrow *) = \beta \cdot (a \leftarrow v)}{(\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \xrightarrow{(t,\ \mathsf{ld},\ a,\ v)}_{\mathsf{TSO}} (\mathsf{pc}', \mathsf{val}[r := v], \mathsf{buf})}$$

$$\frac{\langle instr \rangle = r \leftarrow \texttt{mem}[f_a(\overline{r_a})],\ a = f_a(\mathsf{val}(\overline{r_a})),\ \mathsf{buf}(t)\downarrow(a \leftarrow *) = \varepsilon,\ v = \mathsf{val}(a)}{(\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \xrightarrow{(t,\ \mathsf{ld},\ a,\ v)}_{\mathsf{TSO}} (\mathsf{pc}', \mathsf{val}[r := v], \mathsf{buf})}$$

$$\frac{\langle instr \rangle = \texttt{mem}[f_a(\overline{r_a})] \leftarrow f_v(\overline{r_v}),\ a = f_a(\mathsf{val}(\overline{r_a})),\ v = f_v(\mathsf{val}(\overline{r_v}))}{(\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \xrightarrow{(t,\ \mathsf{isu})}_{\mathsf{TSO}} (\mathsf{pc}', \mathsf{val}, \mathsf{buf}[t := \mathsf{buf}(t) \cdot (a \leftarrow v)])}$$

$$\frac{\mathsf{buf}(t) = (a \leftarrow v) \cdot \beta}{(\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \xrightarrow{(t,\ \mathsf{st},\ a,\ v)}_{\mathsf{TSO}} (\mathsf{pc}, \mathsf{val}[a := v], \mathsf{buf}[t := \beta])}$$

$$\frac{\langle instr \rangle = \texttt{mfence},\ \mathsf{buf}(t) = \varepsilon}{(\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \xrightarrow{(t,\ \mathsf{loc})}_{\mathsf{TSO}} (\mathsf{pc}', \mathsf{val}, \mathsf{buf})}$$

$$\frac{\langle instr \rangle = r \leftarrow f(\overline{r})}{(\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \xrightarrow{(t,\ \mathsf{loc})}_{\mathsf{TSO}} (\mathsf{pc}', \mathsf{val}[r := f(\mathsf{val}(\overline{r}))], \mathsf{buf})}$$

$$\frac{\langle instr \rangle = \texttt{assert}\ f(\overline{r}),\ f(\mathsf{val}(\overline{r})) \neq 0}{(\mathsf{pc}, \mathsf{val}, \mathsf{buf}) \xrightarrow{(t,\ \mathsf{loc})}_{\mathsf{TSO}} (\mathsf{pc}', \mathsf{val}, \mathsf{buf})}$$

instruction generates two actions: an issue followed by the store. Memory fences have no effect under SC. We denote the set of all SC computations of $\mathcal{P}$ by

$$\mathsf{C}_{\mathsf{SC}}(\mathcal{P}) := \{\sigma \in \mathsf{ACT}^* \mid \mathsf{s}_0 \xrightarrow{\sigma}_{\mathsf{SC}} \mathsf{s} \text{ for some SC state } \mathsf{s}\}.$$

3 TSO Robustness

Robustness ensures that the behaviour of a program does not change when it is run on TSO hardware as compared to SC. We study trace-based robustness as in [29,10,11,5,9]. Traces capture the essence of a computation: the control and data dependencies among actions. More formally, consider some computation $\alpha \in \mathsf{C}_{\mathsf{SC}}(\mathcal{P}) \cup \mathsf{C}_{\mathsf{TSO}}(\mathcal{P})$. The *trace* $\mathsf{Tr}(\alpha)$ is a graph where the nodes are labelled by the actions in α (stores and issue yield one node). The arcs are defined by the following relations. We have a per thread $t \in \mathsf{THRD}$ total order $\rightarrow^t_{\mathsf{po}}$ that gives the order in which the actions of t where issued. Similarly, we have a per address $a \in \mathsf{DOM}$ total order $\rightarrow^a_{\mathsf{st}}$ that gives the ordering of all stores to a. We call the unions $\rightarrow_{\mathsf{po}} := \cup_{t \in \mathsf{THRD}} \rightarrow^t_{\mathsf{po}}$ and $\rightarrow_{\mathsf{st}} := \cup_{a \in \mathsf{DOM}} \rightarrow^a_{\mathsf{st}}$ the *program order* and the *store order* of the trace. Finally, there is a *source relation* $\rightarrow_{\mathsf{src}}$ that determines the store from which a load receives its value. By $\mathsf{Tr}_{\mathsf{mm}}(\mathcal{P}) := \mathsf{Tr}(\mathsf{C}_{\mathsf{mm}}(\mathcal{P}))$ with $\mathsf{mm} \in \{\mathsf{SC}, \mathsf{TSO}\}$ we denote the set of all *SC/TSO traces* of program $\mathcal{P}$. The *TSO robustness problem* checks whether the sets coincide.

Given: A parallel program $\mathcal{P}$.
Problem: Does $\mathsf{Tr}_{\mathsf{TSO}}(\mathcal{P}) = \mathsf{Tr}_{\mathsf{SC}}(\mathcal{P})$ hold?

Since inclusion $\mathsf{Tr}_{\mathsf{SC}}(\mathcal{P}) \subseteq \mathsf{Tr}_{\mathsf{TSO}}(\mathcal{P})$ always holds, we only have to check the reverse inclusion. We call a computation $\tau \in \mathsf{C}_{\mathsf{TSO}}(\mathcal{P})$ *violating* if its trace is not among the SC traces of the program, i.e., $\mathsf{Tr}(\tau) \notin \mathsf{Tr}_{\mathsf{SC}}(\mathcal{P})$. Violating TSO-computations employ cyclic accesses to addresses that SC is unable to serialize [29]. These cyclic accesses are made visible using a *conflict relation* from loads to stores. Intuitively, $\mathsf{ld} \rightarrow_{\mathsf{cf}} \mathsf{st}$ means that st overwrites the value that ld reads. The union of all four relations is commonly called *happens-before relation* of the trace, $\rightarrow_{\mathsf{hb}} := \rightarrow_{\mathsf{po}} \cup \rightarrow_{\mathsf{st}} \cup \rightarrow_{\mathsf{src}} \cup \rightarrow_{\mathsf{cf}}$.

Lemma 1 ([29]). *Consider TSO trace $\mathsf{Tr}(\tau) \in \mathsf{Tr}_{\mathsf{TSO}}(\mathcal{P})$. Then $\mathsf{Tr}(\tau) \in \mathsf{Tr}_{\mathsf{SC}}(\mathcal{P})$ iff $\rightarrow_{\mathsf{hb}}$ is acyclic.*

Consider computation τ in Figure 3. The load from thread t_1 conflicts with the store from t_2 because the load reads the initial value of y while the store overwrites it. The situation with the load from t_2 and the store from t_1 is symmetric. Together with the program order, the conflict relations produce a cycle:

$$\mathsf{Tr}(\tau): \quad \begin{array}{ccc} (t_1, \mathsf{st}, \mathsf{x}, 1) & \overset{cf}{\longleftarrow} & (t_2, \mathsf{ld}, \mathsf{x}, 0) \\ \Big\downarrow po & & \Big\downarrow po \\ (t_1, \mathsf{ld}, \mathsf{y}, 0) & \overset{cf}{\longrightarrow} & (t_2, \mathsf{st}, \mathsf{y}, 1) \end{array}$$

Indeed, there is no SC computation with this trace, as predicted by Lemma 1.

Lemma 1 does not provide a method for finding cyclic traces. We have recently shown that TSO robustness is decidable, in fact, PSPACE-complete [9]. The algorithm underlying this result, however, is based on enumeration and not meant to be implemented. The main contribution of the present work is a novel and practical approach to checking robustness.

The only concept we keep from our earlier work are minimal violations. A *minimal violation* is a violating computation that uses a minimal total number of delays. Interestingly, for minimal violations the following holds.

Lemma 2 (Locality [9,8]). *In a minimal violation, only one thread delays stores.*

Consider the computation in Figure 3. It relies on a single delay in thread t_1 and, indeed, is a minimal violation. As predicted by the lemma, the second thread writes to its buffer and immediately flushes it.

4 Attacks on TSO Robustness

Our approach to checking TSO robustness combines two insights. We first rephrase robustness in terms of a simpler problem: the absence of feasible attacks. We then devise an algorithm that checks attacks for feasibility. Interestingly, SC reachability techniques are sufficient for this purpose. Together, this yields a sound and complete reduction of TSO robustness to SC reachability.

The notion of attacks is inspired by the shape of minimal violations. We show that if a program is not robust, then there are violations of the form shown in

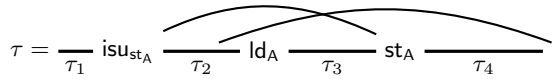

Fig. 4. TSO witness for the attack $(\mathsf{t_A}, \mathsf{stinst}, \mathsf{ldinst})$. It satisfies the following constraints. **(W1)** Only the attacker delays stores. **(W2)** Store $\mathsf{st_A}$ is an instance of stinst. It is the first store of the attacker that is delayed. Load $\mathsf{ld_A}$ is an instance of ldinst. It is the last action of the attacker that is overstepped by $\mathsf{st_A}$. So τ_2 contains loads, assignments, asserts, and issues, but no fences and stores of the attacker. It may contain arbitrary helper actions. **(W3)** We require $\mathsf{ld_A} \rightarrow_{\mathsf{hb}}^{+} \mathsf{act}$ for every action act in $\mathsf{ld_A} \cdot \tau_3 \cdot \mathsf{st_A}$. An issue + store of a helper is counted as one action act. **(W4)** Sequence τ_4 only consists of stores of the attacker that were issued before $\mathsf{ld_A}$ and that have been delayed. **(W5)** All these stores st satisfy $addr(\mathsf{st}) \neq addr(\mathsf{ld_A})$, i.e., $\mathsf{ld_A}$ has not read its value early.

Figure 4: one thread, the *attacker*, delays a store action $\mathsf{st_A}$ past a later load action $\mathsf{ld_A}$ in order to break robustness. The remaining threads become *helpers* and provide a happens-before path from $\mathsf{ld_A}$ to $\mathsf{st_A}$. This yields a happens-before cycle and shows non-robustness.

Thread, store instruction stinst of $\mathsf{st_A}$, and load instruction ldinst of $\mathsf{ld_A}$ are syntactic objects. The idea of our approach is to fix these three parameters, the *attack*, prior to the analysis. The algorithm then tries to find a witness computation that proves the attack feasible.

Definition 1. *An* attack $A = (t_A, \mathit{stinst}, \mathit{ldinst})$ *consists of a thread* $t_A \in \mathit{THRD}$ *called* attacker*, a store instruction* stinst *and a load instruction* ldinst*. A* TSO witness for A *is a computation of the form in Figure 4, i.e., it satisfies* **(W1)** *to* **(W5)***. If a TSO witness exists, the attack is called* feasible*.*

In Dekker's algorithm, there is an attack $\mathsf{A} = (\mathsf{t_A}, \mathsf{stinst}, \mathsf{ldinst})$ with $\mathsf{t_A} = t_1$, stinst the store at l_0, and ldinst the load at l_1. A TSO witness for this attack is the computation τ from Figure 3. With reference to Figure 4, we have $\tau_1 = \varepsilon$, $\mathsf{isu_{st_A}} = (t_1, \mathsf{isu})$, $\tau_2 = \varepsilon$, $\mathsf{ld_A} = (t_1, \mathsf{ld}, \mathsf{y}, 0)$, $\tau_3 = (t_2, \mathsf{isu}) \cdot (t_2, \mathsf{st}, \mathsf{y}, 1) \cdot (t_2, \mathsf{ld}, \mathsf{x}, 0)$, $\mathsf{st_A} = (t_1, \mathsf{st}, \mathsf{x}, 1)$, $\tau_4 = \varepsilon$. The program also contains a symmetric attack $\mathsf{A'}$ with t_2 as the attacker.

Although TSO witnesses are quite restrictive computations, robustness can be reduced to verifying that no attack has a TSO witness.

Theorem 1 (Complete Characterization of Robustness with Attacks). *Program* $\mathcal{P}$ *is robust iff no attack is feasible, i.e., no attack admits a TSO witness.*

Proof. The existence of a TSO witness implies non-robustness of the program. Indeed, a TSO witness comes with a happens-before cycle $\mathsf{st_A} \rightarrow_{\mathsf{po}}^{+} \mathsf{ld_A} \rightarrow_{\mathsf{hb}}^{+} \mathsf{st_A}$. We argue that also the reverse holds: if a program is not robust, there is a feasible attack. Assume $\mathcal{P}$ is not robust. We construct a TSO witness computation. Among the violating computations, we select $\tau \in \mathsf{C_{TSO}}(\mathcal{P})$ where the number of delays is minimal. The computation need not be unique. By Lemma 2, in τ only one thread $\mathsf{t_A}$ uses its buffer and **(W1)** holds. We elaborate on the shape of τ.

Initially, the attacker executes under SC so that stores immediately follow their issues. This computation is embedded into τ_1 in Figure 4. Eventually, the attacker starts delaying stores. Let st_A be the first store that is delayed. It gets reordered past several loads, the last of which being ld_A. This shows **(W2)**.

Consider the helper actions in τ_3. To see that we can assume **(W3)**, first note that $\mathsf{ld}_\mathsf{A} \rightarrow^+_{\mathsf{hb}} \mathsf{st}_\mathsf{A}$ holds. If there was no such path, st_A could be placed before ld_A without changing the trace. This would save a delay, in contradiction to minimality of τ. Assume $\tau_3 = \tau_3' \cdot \mathsf{act} \cdot \tau_3''$ where act is the first action so that $\mathsf{ld}_\mathsf{A} \not\rightarrow^+_{\mathsf{hb}} \mathsf{act}$. Then act is independent from all actions in $\mathsf{ld}_\mathsf{A} \cdot \tau_3'$. We find a computation with the same trace where act is placed before ld_A.

With cycle $\mathsf{st}_\mathsf{A} \rightarrow^+_{\mathsf{po}} \mathsf{ld}_\mathsf{A} \rightarrow^+_{\mathsf{hb}} \mathsf{st}_\mathsf{A}$, computation τ_4 only needs to contain the stores of the attacker that have been delayed past ld_A. Since these stores are non-blocking, the helpers can stop with the last action in τ_3. We can moreover assume ld_A to be the program order last action of the attacker. **(W4)** holds.

We now argue that ld_A has not read its value early from any of the delayed stores, **(W5)**. Towards a contradiction, assume ld_A obtained its value from st in $\tau_4 = \tau_{41} \cdot \mathsf{st} \cdot \tau_{42}$. There is a computation τ' where we avoid the early read: it replaces τ_4 by $\tau_{41} \cdot \mathsf{st} \cdot \mathsf{ld}_\mathsf{A} \cdot \tau_{42}$. The traces of τ and τ' coincide, but τ' saves the delay of st past ld_A. A contradiction to minimality.

It is readily checked that τ is a TSO witness for the attack $(\mathsf{t}_\mathsf{A}, \mathsf{stinst}, \mathsf{ldinst})$ where stinst and ldinst are the instructions that st_A and ld_A are derived from. □

Since the number of attacks is only quadratic in the size of the program, we can just enumerate them and check whether one admits a TSO witness. To check whether a witness exists, we employ the instrumentation described in the following section.

5 Instrumentation

Consider program $\mathcal{P}$ with attack $\mathsf{A} = (\mathsf{t}_\mathsf{A}, \mathsf{stinst}, \mathsf{ldinst})$. TSO witnesses for A only make limited use of the store buffers, to an extent that allows us to characterize them by SC computations in a program $\mathcal{P}_\mathsf{A}$ that is *instrumented for attack A*. By instrumentation we mean that $\mathcal{P}_\mathsf{A}$ replaces every thread by a modified version. Capturing TSO witnesses with a program that runs under SC is difficult for two reasons. First, TSO has unbounded store buffers which can delay arbitrarily many stores. Second, the happens-before dependence that the helpers create may involve an arbitrary number of actions. Our instrumentation copes with both problems, using the following tricks.

To handle store buffering, we instrument the attacker thread (Section 5.1). Essentially, we emulate store buffering under SC using auxiliary addresses. To explain the idea, consider the TSO witness in Figure 4. When the instrumented attacker executes the delayed stores $\mathsf{st}_\mathsf{A} \cdot \tau_4$ under SC, they occur right behind their issue actions. To mimic store buffering, these stores now access auxiliary addresses that the other threads do not load. As a result, the stores remain invisible to the helpers. This is as intended: the delayed stores $\mathsf{st}_\mathsf{A} \cdot \tau_4$ in Figure 4 are also never accessed by helper threads. But how many auxiliary addresses do we need to faithfully simulate buffers? It is sufficient to have *a single auxiliary*

address per address in the program. The reason is that a load always reads the most recent store to its address that is held in the buffer.

To build up a happens-before path from ld_A to st_A, we instrument the helper threads (Section 5.2). The question is how to decide whether a new action act is in happens-before relation with an earlier action act' so that $\mathsf{ld}_\mathsf{A} \rightarrow_{\mathsf{hb}}^* \mathsf{act}' \rightarrow_{\mathsf{hb}}^* \mathsf{act}$. What is the information we need about the earlier actions in order to append act? It is sufficient to know two facts. Has the thread already contributed an action act'? This information ensures $\mathsf{act}' \rightarrow_{\mathsf{po}}^* \mathsf{act}$, and can be kept in the control flow of the thread. Moreover, we keep track of whether the path contains a load or store access to the address $addr(\mathsf{act})$. If there was a load access $\mathsf{act}' = \mathsf{ld}$, we can add a store $\mathsf{act} = \mathsf{st}$ and get $\mathsf{ld} \rightarrow_{\mathsf{hb}}^* \mathsf{st}$. If there was a store, we are free to add a load or a store. Hence, we need *one auxiliary address* per address in the program for this access information: no access, load access, store access.

Consider the TSO witness for Dekker given in Figure 3. Instead of buffering $(t_1, \mathsf{st}, \mathsf{x}, 1)$, the instrumentation immediately executes the store after its issue action. But instead of address x, the action accesses the auxiliary address (x, d) that is invisible to the other threads. So, the SC computation of the instrumented program roughly looks like this:

$$(t_1, \mathsf{isu}) \cdot (t_1, \mathsf{st}, (\mathsf{x}, \mathsf{d}), 1) \cdot (t_1, \mathsf{ld}, \mathsf{y}, 0) \overset{(1)}{\cdot} (t_2, \mathsf{isu})(t_2, \mathsf{st}, \mathsf{y}, 1) \overset{(2)}{\cdot} (t_2, \mathsf{ld}, \mathsf{x}, 0).$$

At moment (1), we know that there has been a load access to address y. At moment (2), address y has even seen a store. At the end of the computation, address y has seen a store and address x has seen a load.

The store of t_2 can be appended since it is in happens-before relation with the attacker's load. The following load can be added as t_2 has contributed the previous store. The search terminates here since the helper's load accesses address x that was used by the store from the attack.

5.1 Instrumentation of the Attacker

The instrumentation emulates the buffering of stores in a TSO witness (Figure 4). Starting from st_A, the stores are replaced by stores $\mathsf{st}_\mathsf{A}^{\mathsf{aux}}$ to auxiliary addresses (a, d) that are only visible to the attacker. As long as a has not been written, (a, d) holds the initial value 0. Once the attacker stores v into a, we set $\mathtt{mem}[(a, \mathsf{d})] = (v, \mathsf{d})$. In this way, (a, d) always holds the most recent store. A load $r \leftarrow \mathtt{mem}[a]$ of the attacker reads a value v from the buffer whenever $\mathtt{mem}[(a, \mathsf{d})] = (v, \mathsf{d})$; otherwise $\mathtt{mem}[(a, \mathsf{d})] = 0$ and the load obtains the value $v = \mathtt{mem}[a]$ from memory. We turn to the translation.

Let t_A declare registers r^*, have initial location l_0, and define instructions $\langle linst \rangle^*$ that contain stinst and ldinst from the attack. The instrumentation is

$$[\![\mathsf{t}_\mathsf{A}]\!] := \mathtt{thread}\ \tilde{\mathsf{t}_\mathsf{A}}\ \mathtt{regs}\ r^*\ \mathtt{init}\ \ \mathsf{l}_0 \\ \mathtt{begin}\ \langle linst \rangle^*\ [\![\mathsf{stinst}]\!]_{\mathsf{A}1}\ [\![\mathsf{ldinst}]\!]_{\mathsf{A}1}\ [\![\langle linst \rangle]\!]_{\mathsf{A}2}^*\ \mathtt{end}.$$

It introduces a copy of the source code $[\![\langle linst \rangle]\!]_{\mathsf{A}2}^*$ where the stores are replaced by accesses to the auxiliary addresses. To move to the code copy, the attacker uses an instrumented version $[\![\mathsf{stinst}]\!]_{\mathsf{A}1}$ of stinst.

$$[\![\mathsf{l}_1 : \texttt{mem[}e_1\texttt{]} \leftarrow e_2\texttt{; goto } \mathsf{l}_2\texttt{;}]\!]_{\mathsf{A1}} := \mathsf{l}_1 : \texttt{mem[}(e_1, \mathsf{d})\texttt{]} \leftarrow (e_2, \mathsf{d})\texttt{; goto } \tilde{\mathsf{l}}_x\texttt{;} \quad (1)$$
$$\tilde{\mathsf{l}}_x : \texttt{mem[}a_{\mathsf{st_A}}\texttt{]} \leftarrow e_1\texttt{; goto } \tilde{\mathsf{l}}_2\texttt{;}$$

$$[\![\mathsf{l}_1 : r \leftarrow \texttt{mem[}e\texttt{]; goto } \mathsf{l}_2\texttt{;}]\!]_{\mathsf{A1}} := \tilde{\mathsf{l}}_1 : \texttt{assert mem[}(e, \mathsf{d})\texttt{]} = 0\texttt{; goto } \tilde{\mathsf{l}}_{x1}\texttt{;} \quad (2)$$
$$\tilde{\mathsf{l}}_{x1} : \texttt{mem[hb]} \leftarrow \mathsf{true}\texttt{; goto } \tilde{\mathsf{l}}_{x2}\texttt{;}$$
$$\tilde{\mathsf{l}}_{x2} : \texttt{mem[}(e, \mathsf{hb})\texttt{]} \leftarrow \mathsf{lda}\texttt{; goto } \tilde{\mathsf{l}}_{x3}\texttt{;}$$

$$[\![\mathsf{l}_1 : \texttt{mem[}e_1\texttt{]} \leftarrow e_2\texttt{; goto } \mathsf{l}_2\texttt{;}]\!]_{\mathsf{A2}} := \tilde{\mathsf{l}}_1 : \texttt{mem[}(e_1, \mathsf{d})\texttt{]} \leftarrow (e_2, \mathsf{d})\texttt{; goto } \tilde{\mathsf{l}}_2\texttt{;} \quad (3)$$

$$[\![\mathsf{l}_1 : r \leftarrow \texttt{mem[}e\texttt{]; goto } \mathsf{l}_2\texttt{;}]\!]_{\mathsf{A2}} := \tilde{\mathsf{l}}_1 : \texttt{assert mem[}(e, \mathsf{d})\texttt{]} = 0\texttt{; goto } \tilde{\mathsf{l}}_{x1}\texttt{;} \quad (4)$$
$$\tilde{\mathsf{l}}_{x1} : r \leftarrow \texttt{mem[}e\texttt{]; goto } \tilde{\mathsf{l}}_2\texttt{;}$$
$$\tilde{\mathsf{l}}_1 : \texttt{assert mem[}(e, \mathsf{d})\texttt{]} \neq 0\texttt{; goto } \tilde{\mathsf{l}}_{x2}\texttt{;}$$
$$\tilde{\mathsf{l}}_{x2} : (r, \mathsf{d}) \leftarrow \texttt{mem[}(e, \mathsf{d})\texttt{]; goto } \tilde{\mathsf{l}}_2\texttt{;}$$

$$[\![\mathsf{l}_1 : \textit{local}\texttt{; goto } \mathsf{l}_2\texttt{;}]\!]_{\mathsf{A2}} := \tilde{\mathsf{l}}_1 : \textit{local}\texttt{; goto } \tilde{\mathsf{l}}_2\texttt{;} \quad (5)$$

$$[\![\mathsf{l}_1 : \texttt{mfence; goto } \mathsf{l}_2\texttt{;}]\!]_{\mathsf{A2}} := \quad (6)$$

Fig. 5. Instrumentation of the attacker

The translation of instructions is defined in Figure 5. We make a few remarks. The instrumentation (1) of stinst saves the address used in the store in a fresh address $a_{\mathsf{st_A}}$. For the sake of readability, Equation (4) uses memory accesses in asserts. Equation (6) deletes fences, as they forbid to delay $\mathsf{st_A}$ over $\mathsf{ld_A}$. Equation (2) checks that the load used in the attack has not read its value early, sets an auxiliary happens-before address (e, hb) to access level load, lda, and halts the attacker. We postpone the definition of access levels until the translation of helpers. The equation also sets a flag hb that forbids helpers to execute actions not contributing to the happens-before path. Figure 6 illustrates the instrumentation on our running example.

5.2 Instrumentation of Helpers

In TSO witnesses, all helper actions after $\mathsf{ld_A}$ are in happens-before relation with $\mathsf{ld_A}$, by **(W3)**. To ensure this, we use Lemma 3. The proof from left to right is by definition of happens-before. For the reverse direction, note that happens-before is *stable under insertion*. Consider $\mathsf{st} \rightarrow_{\mathsf{src}} \mathsf{ld}$. A happens-before relation remains valid in any computation that places actions between st and ld.

Lemma 3. *Consider $\tau = \tau_1 \cdot \mathsf{act}_1 \cdot \tau_2 \in C_{SC}(\mathcal{P})$ where for all act_2 in τ_2 we have $\mathsf{act}_1 \rightarrow^*_{\mathsf{hb}} \mathsf{act}_2$. Computation $\tau \cdot \mathsf{act}$ satisfies $\mathsf{act}_1 \rightarrow^*_{\mathsf{hb}} \mathsf{act}$ iff*
(i) *there is an action act_2 in $\mathsf{act}_1 \cdot \tau_2$ with $\mathit{thread}(\mathsf{act}_2) = \mathit{thread}(\mathsf{act})$ or*
(ii) *act is a load whose address is stored in $\mathsf{act}_1 \cdot \tau_2$ or*
(iii) *act is a store (with issue) whose address is loaded or stored in $\mathsf{act}_1 \cdot \tau_2$.*

The lemma suggests the following instrumentation. For every helper t, we track whether it has executed an action that depends on $\mathsf{ld_A}$. The idea is to use the control flow. Upon detection of this first action, the thread moves to a copy of its code. All actions from this copy stay in happens-before relation with $\mathsf{ld_A}$.

```
thread t̃1 regs r1 init l0 begin
/* Original code */
l0: mem[x] ← 1; goto l1;
l1: r1 ← mem[y]; goto l2;

/* Instrumented stinst */
l0: mem[(x,d)] ← (1,d); goto l̃x;
l̃x: mem[a_stA] ← x; goto l̃1;

/* Instrumented ldinst */
l̃1: assert mem[(y,d)] = 0; goto l̃x1;
l̃x1: mem[hb] ← true; goto l̃x2;
l̃x2: mem[(y,hb)] ← lda; goto l̃x3;
end

/* Instrumented copy of the store */
l̃0: mem[(x,d)] ← (1,d); goto l̃1;

/* Instrumented copy of the load */
l̃1: assert mem[(y,d)] = 0; goto l̃x4;
l̃x4: r1 ← mem[y]; goto l̃2;
l̃1: assert mem[(y,d)] ≠ 0; goto l̃x5;
l̃x5: (r1,d) ← mem[(y,d)]; goto l̃2;
```

Fig. 6. Attacker instrumentation of thread t_1 in Dekker from Figure 2. The attack's store is the store at label l_0, the load is the load at label l_1.

It remains to decide whether an action act allows a thread to move to the code copy. According to Lemma 3, this depends on the earlier accesses to the address $a = addr(\mathsf{act})$. We introduce auxiliary *happens-before addresses* (a, hb) that provide this access information. The addresses (a, hb) range over the domain $\{0, \mathsf{lda}, \mathsf{sta}\}$ of *access types*. It is sufficient to keep track of the *maximal* access type wrt. the ordering 0 (no access) $<$ lda (load access) $<$ sta (store access).

For the definition, consider a helper thread t that declares registers r^*, has initial label l_0, and defines instructions $\langle linst \rangle^*$. The instrumented thread is

$$[\![t]\!] := \texttt{thread}\ \tilde{t}\ \texttt{regs}\ \tilde{r}, r^*\ \texttt{init}\ \mathsf{l}_0 \\ \texttt{begin}\ [\![\langle linst \rangle]\!]^*_{\mathsf{H0}}\ [\![\langle ldstinst \rangle]\!]^*_{\mathsf{H1}}\ [\![\langle linst \rangle]\!]^*_{\mathsf{H2}}\ [\![\langle \mathsf{l} \rangle]\!]^*_{\mathsf{H3}}\ \texttt{end}.$$

The instrumentation of the original code $[\![\langle linst \rangle]\!]_{\mathsf{H0}}$ forces helpers to either enter the code copy or stop when the hb flag is raised. To move to the code copy, we instrument the subsequence $\langle ldstinst \rangle^*$ of all load and store instructions in $\langle linst \rangle^*$. The code copy is $[\![\langle linst \rangle]\!]^*_{\mathsf{H2}}$. Let $\langle \mathsf{l} \rangle^*$ be all labels used by the thread. The additional instructions $[\![\langle \mathsf{l} \rangle]\!]^*_{\mathsf{H3}}$ raise a success flag when a TSO witness has been found.

The translation of instructions is given in Figure 7. We make some remarks. Transitions to the code copy check the auxiliary addresses for whether the current action is in happens-before relation with ld_A. Loads in Equation (8) check for an earlier store access to their address, Lemma 3(ii). Stores in Equation (9) require that the address has seen at least a load, Lemma 3(iii). They set the access level to sta. Loads and stores in the code copy maintain the auxiliary addresses to contain the maximal access types, Equations (12) and (11). Note the auxiliary register $\tilde{r}$ that ensures we do not overwrite the address. At every label of the code copy we add a check, Equation (13), whether the address used in the attack's store has been accessed in the code copy. If so, a success flag is raised.

$$[\![\mathsf{l}_1\colon \mathit{instr};\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{H0}} := \mathsf{l}_1\colon \mathtt{assert}\ \mathtt{mem}[\mathsf{hb}] = 0;\ \mathtt{goto}\ \mathsf{l}_x; \quad (7)$$
$$\mathsf{l}_x\colon \mathit{instr};\ \mathtt{goto}\ \mathsf{l}_2;$$

$$[\![\mathsf{l}_1\colon r \leftarrow \mathtt{mem}[e];\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{H1}} := \mathsf{l}_1\colon \mathtt{assert}\ \mathtt{mem}[(e,\mathsf{hb})] = \mathsf{sta};\ \mathtt{goto}\ \tilde{\mathsf{l}}_x; \quad (8)$$
$$\tilde{\mathsf{l}}_x\colon r \leftarrow \mathtt{mem}[e];\ \mathtt{goto}\ \tilde{\mathsf{l}}_2;$$

$$[\![\mathsf{l}_1\colon \mathtt{mem}[e_1] \leftarrow e_2;\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{H1}} := \mathsf{l}_1\colon \mathtt{assert}\ \mathtt{mem}[(e_1,\mathsf{hb})] \geq \mathsf{lda};\ \mathtt{goto}\ \tilde{\mathsf{l}}_{x1}; \quad (9)$$
$$\tilde{\mathsf{l}}_{x1}\colon \mathtt{mem}[e_1] \leftarrow e_2;\ \mathtt{goto}\ \tilde{\mathsf{l}}_{x2};$$
$$\tilde{\mathsf{l}}_{x2}\colon \mathtt{mem}[(e_1,\mathsf{hb})] \leftarrow \mathsf{sta};\ \mathtt{goto}\ \tilde{\mathsf{l}}_2;$$

$$[\![\mathsf{l}_1\colon \mathit{local/mfence};\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{H2}} := \tilde{\mathsf{l}}_1\colon \mathit{local/mfence};\ \mathtt{goto}\ \tilde{\mathsf{l}}_2; \quad (10)$$

$$[\![\mathsf{l}_1\colon \mathtt{mem}[e_1] \leftarrow e_2;\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{H2}} := \tilde{\mathsf{l}}_1\colon \mathtt{mem}[e_1] \leftarrow e_2;\ \mathtt{goto}\ \tilde{\mathsf{l}}_x; \quad (11)$$
$$\tilde{\mathsf{l}}_x\colon \mathtt{mem}[(e_1,\mathsf{hb})] \leftarrow \mathsf{sta};\ \mathtt{goto}\ \tilde{\mathsf{l}}_2;$$

$$[\![\mathsf{l}_1\colon r \leftarrow \mathtt{mem}[e];\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{H2}} := \tilde{\mathsf{l}}_1\colon \tilde{r} \leftarrow e;\ \mathtt{goto}\ \tilde{\mathsf{l}}_{x1}; \quad (12)$$
$$\tilde{\mathsf{l}}_{x1}\colon r \leftarrow \mathtt{mem}[\tilde{r}];\ \mathtt{goto}\ \tilde{\mathsf{l}}_{x2};$$
$$\tilde{\mathsf{l}}_{x2}\colon \mathtt{mem}[(\tilde{r},\mathsf{hb})] \leftarrow \mathsf{max}\{\mathsf{lda}, \mathtt{mem}[(\tilde{r},\mathsf{hb})]\};\ \mathtt{goto}\ \tilde{\mathsf{l}}_2;$$

$$[\![\mathsf{l}]\!]_{\mathsf{H3}} := \tilde{\mathsf{l}}\colon \tilde{r} \leftarrow \mathtt{mem}[a_{\mathsf{st_A}}];\ \mathtt{goto}\ \tilde{\mathsf{l}}_{x1}; \quad (13)$$
$$\tilde{\mathsf{l}}_{x1}\colon \tilde{r} \leftarrow \mathtt{mem}[(\tilde{r},\mathsf{hb})];\ \mathtt{goto}\ \tilde{\mathsf{l}}_{x2};$$
$$\tilde{\mathsf{l}}_{x2}\colon \mathtt{assert}\ \tilde{r} \neq 0;\ \mathtt{goto}\ \tilde{\mathsf{l}}_{x3};$$
$$\tilde{\mathsf{l}}_{x3}\colon \mathtt{mem}[\mathsf{suc}] \leftarrow \mathsf{true};\ \mathtt{goto}\ \tilde{\mathsf{l}}_{x4};$$

Fig. 7. Instrumentation of helpers

5.3 Soundness and Completeness

The flag indicates that the SC computation corresponds to a TSO witness, and we call $(\mathsf{pc}, \mathsf{val})$ with $\mathsf{val}(\mathsf{suc}) = \mathsf{true}$ a *goal configuration*. The instrumentation thus reduces feasibility of attack A to SC reachability of a goal configuration in program $\mathcal{P}_\mathsf{A}$. The instrumentation is sound and complete. If a goal configuration is reachable, we can reconstruct a TSO witness for the attack. In turn, every TSO witness ensures the goal configuration is reachable.

Theorem 2 (Soundness and Completeness [8]). *Attack A is feasible in program $\mathcal{P}$ iff program $\mathcal{P}_A$ reaches a goal configuration under SC.*

In combination with Theorem 1, we can check robustness by inspecting all $\mathcal{P}_\mathsf{A}$.

Theorem 3 (From TSO Robustness to SC Reachability). *Program $\mathcal{P}$ is robust iff no instrumentation $\mathcal{P}_A$ reaches a goal configuration under SC.*

The instrumentation we provide is linear in size. Then, it follows from Theorem 3 that checking robustness for programs over finite data domains is in PSpace. The problem is actually PSpace-complete due to the lower bound in [9].

6 TSO Robustness for Parameterized Programs

We extend the study of robustness to *parameterized programs*. A parameterized program represents an infinite family of instance programs that replicate the

threads multiple times. Syntactically, parameterized programs coincide with the parallel programs we introduced in Section 2: they have a name and declare a finite set of threads $t_1, \ldots, t_k$. The difference is in the semantics. A parameterized program defines, for every vector $I = (n_1, \ldots, n_k) \in \mathbb{N}^k$, an *instance program* $\mathcal{P}(I)$ that declares n_i copies of thread t_i.

In the parameterized setting, the robustness problem asks for whether all instances of a given program are robust:

Given: A parameterized program $\mathcal{P}$.
Problem: Does $\mathsf{Tr}_{\mathsf{TSO}}(\mathcal{P}(I)) = \mathsf{Tr}_{\mathsf{SC}}(\mathcal{P}(I))$ hold for all instances $\mathcal{P}(I)$ of $\mathcal{P}$?

The problem is interesting because libraries usually cannot make assumptions on the number of threads that use their functions. They have to guarantee proper functioning for any number.

We reduce robustness for parameterized programs to a parameterized version of reachability, based on the following insight. A parameterized program is not robust if and only if there is an instance $\mathcal{P}(I)$ that is not robust. With Theorem 1, instance $\mathcal{P}(I)$ is not robust if and only if there is an attack A that is feasible. With the instrumentation from Section 5 and Theorem 2, this feasibility can be checked as reachability of a goal configuration in $\mathcal{P}(I)_{\mathsf{A}}$.

Algorithmically, it is impossible to instrument all (infinitely many) instance programs. Instead, the idea is to instrument directly the parameterized program $\mathcal{P}$ towards the attack A. Using the constructions from Section 5, we modify every thread and again obtain program $\mathcal{P}_{\mathsf{A}}$, which is now parameterized.

Actually, for the attacker we have to be slightly more careful. In an instance program, only one copy of the thread should act as the attacker, the remaining copies must behave like helpers. Therefore, we instrument the thread not only as an attacker, but also as a helper. To ensure that only one copy of the attacker delays stores, we add an additional flag variable. Before starting an attack, the thread checks this variable. If it contains the initial value, the thread sets the flag and starts delaying stores. If it has a different value, the thread continues to run sequentially. This check requires an atomic test-and-set operation which can be implemented on x86 by the `lock cmpxchg` instruction. Support for locked instructions is immediate to add to our programming model.

Modulo these two changes, the instances $\mathcal{P}_{\mathsf{A}}(I)$ coincide with the instrumentations $\mathcal{P}(I)_{\mathsf{A}}$. Together with the previous argumentation this justifies the following theorem.

Theorem 4. *A parameterized program $\mathcal{P}$ is not robust iff there is an attack A so that an instance $\mathcal{P}_{\mathsf{A}}(I)$ of program $\mathcal{P}_{\mathsf{A}}$ reaches a goal configuration under SC.*

Reachability of a goal configuration in one instance of $\mathcal{P}_{\mathsf{A}}$ can be reformulated as a coverability problem for Petri nets, which is known to be decidable [27]. The key observation in the reduction to Petri nets is that threads in instance programs never use their identifiers, simply because they are copies of the same source code. This means there is no need to track the identity of threads, it is sufficient to count how many instances of a thread are in each state — a technique known as counter abstraction [14].

Theorem 5 ([8]). *Robustness for parameterized programs over finite data domains is decidable and* EXPSPACE*-hard — already for Boolean programs.*

For the lower bound, we in turn encode the coverability problem for Petri nets into robustness for parameterized programs [24].

7 Fence Insertion

To ease the presentation, we return to parallel programs. Since the algorithm only relies on a robustness checker, it carries over to the parametric setting.

Our goal is to insert a set of fences that ensure robustness of the resulting program. By *inserting a fence at label* l we mean the following modification of the program. Introduce a fresh label l_f. Then, translate each instruction l: inst; `goto` l'; into l_f: inst; `goto` l';. Finally, add an instruction l: `mfence; goto` l_f;.

We call a set of labels $\mathcal{F}$ in program $\mathcal{P}$ a *valid fence set* if inserting fences at these labels yields a robust program. We say that $\mathcal{F}$ is *irreducible* if no strict subset is a valid fence set. In general, however, we look for a valid fence set which is *optimal* in some sense, and pose the *fence computation problem* as follows:

Given: A program $\mathcal{P}$ and a strictly positive *cost function* $\mathcal{C}\colon \mathsf{LAB} \to \mathbb{R}^+$.
Problem: Compute a valid fence set $\mathcal{F}$ with $\Sigma_{\mathsf{l}\in\mathcal{F}}\mathcal{C}(\mathsf{l})$ minimal.

Since we assume $\mathcal{C}$ to be strictly positive, every optimal fence set is irreducible.

We consider two criteria of optimality: minimization of program size and maximization of program performance. By solving the problem for $\mathcal{C} \equiv 1$ we compute a fence set of minimal size, thus minimizing the code size of the fenced program. Maximization of program performance requires minimizing the number of times memory fence instructions are executed: practical measurements [1] show that it is impossible to save CPU cycles by executing more fences, but with less stores in the TSO buffer. For this, $\mathcal{C}(\mathsf{l})$ is defined as the frequency at which instructions labeled by l occur in executions of the original program $\mathcal{P}$. Concrete values of $\mathcal{C}$ can be either estimated by profiling or computed by mathematical reasoning about the program.

From a complexity point of view, fence computation is at least as hard as robustness. Indeed, robustness holds if and only if the optimal valid fence set is $\mathcal{F} = \emptyset$. Actually, since fence sets can be enumerated, computing an optimal valid fence set does not require more space than checking robustness. Notice that this also holds in the parameterized case.

Theorem 6. *For programs over finite domains, fence computation is* PSPACE*-complete. In the parameterized case, it is decidable and* EXPSPACE*-hard.*

In the remainder of the section, we give a practical algorithm for computing optimal valid fence sets.

7.1 Fence Sets for Attacks

We say that a label l is *involved in the attack* $\mathsf{A} = (\mathsf{t}_\mathsf{A}, \mathsf{stinst}, \mathsf{ldinst})$ if it belongs to some path in the control flow graph of t_A from the destination label of stinst to the source label of ldinst. We denote the set of all such labels by $\mathcal{L}_\mathsf{A}$.

We call a set of labels $\mathcal{F}_\mathsf{A}$ *an eliminating fence set for attack* A if adding fences at all labels in $\mathcal{F}_\mathsf{A}$ eliminates the attack. Dekker's algorithm has two eliminating fence sets: $\mathcal{F}_\mathsf{A} = \{l_1\}$ eliminates the only attack by t_1, and $\mathcal{F}_{\mathsf{A}'} = \{l_1'\}$ eliminates the only attack by t_2. Actually, the sets are *irreducible*: no strict subset eliminates the attack. Note that any irreducible eliminating set $\mathcal{F}_\mathsf{A}$ satisfies $\mathcal{F}_\mathsf{A} \subseteq \mathcal{L}_\mathsf{A}$.

Lemma 4. *Every irreducible valid fence set $\mathcal{F}$ can be represented as a union of irreducible eliminating fence sets for all feasible attacks.*

Proof. By Theorem 1, fence set $\mathcal{F}$ eliminates all feasible attacks. Therefore, it includes some irreducible eliminating fence set $\mathcal{F}_\mathsf{A}$ for every feasible attack A. By irreducibility, $\mathcal{F}$ cannot contain labels outside the union of these $\mathcal{F}_\mathsf{A}$ sets. □

In compliance with the lemma, in Dekker's program $\mathcal{F} = \mathcal{F}_\mathsf{A} \cup \mathcal{F}_{\mathsf{A}'}$.

Lemma 4 is useful for fence computation since optimal fence sets are always irreducible. All irreducible eliminating fence sets for attacks can be constructed by an exhaustive search through all selections of labels involved in the attack.

Note that this search may raise an exponential number of reachability queries. In practice this rarely constitutes a problem. First, attacks seldom involve a large number of labels, so the number of candidates is small. Second, the reachability checks can be avoided if a candidate fence set covers all the ways in the control flow graph from stinst to ldinst.

7.2 Computing an Optimal Valid Fence Set

To choose among the sets $\mathcal{F}_\mathsf{A}$, we set up a 0/1-integer linear programming (ILP) problem $M_\mathcal{P} \cdot x_\mathcal{P} \geq b_\mathcal{P}$. The optimal solutions $f(x_\mathcal{P}) \to \min$ correspond to optimal fence sets. Here, 0/1 means the variables are Boolean.

We define inequalities that encode the feasible attacks with their corrections. Consider attack A for which we have determined the irreducible eliminating fence sets $\mathcal{F}_1, \ldots, \mathcal{F}_n$. For each set, we introduce a variable $x_{\mathcal{F}_i}$ and set up Inequality (14)(left). It selects a fence set to eliminate the attack.

$$\sum_{1 \leq i \leq n} x_{\mathcal{F}_i} \geq 1 \qquad \sum_{\mathsf{l} \in \mathcal{F}_i} x_\mathsf{l} \geq |\mathcal{F}_i| x_{\mathcal{F}_i} \qquad f(x_\mathcal{P}) := \sum_{\mathsf{l} \in \mathsf{LAB}} \mathcal{C}(\mathsf{l}) x_\mathsf{l}. \tag{14}$$

When $\mathcal{F}_i$ has been chosen, we insert a fence at each of its labels l. We add further variables x_l, and encode this insertion by Inequality (14)(center). By definition of the ILP, the variables $x_{\mathcal{F}_i}$ and x_l will only take Boolean values 0 or 1. So if $x_{\mathcal{F}_i}$ is set to 1, the inequality requires that all x_l with $\mathsf{l} \in \mathcal{F}_i$ are set to 1.

Our goal is to select fences with minimal costs. We encode this into the objective function (14)(right). An optimal solution x^* of the resulting 0/1-ILP denotes the fence set $\mathcal{F}(x^*) := \{\mathsf{l} \in \mathsf{LAB} \mid x^*_\mathsf{l} = 1\}$.

Theorem 7. *$\mathcal{F}(x^*)$ is valid and optimal, and thus solves fence computation.*

8 Experimental Evaluation

We implemented our algorithms in a prototype called Trencher [1]. The tool performs the reduction of robustness to SC reachability given in Section 5 and computes a minimal fence set as described in Section 7. Trencher executes independent reachability queries in parallel and uses SPIN [17] as back-end model checker. With Trencher, we have performed a series of experiments.

8.1 Examples

The first class of examples are mutual exclusion protocols that are implemented via shared variables. These protocols are typically not robust under TSO and require additional fences after stores to synchronization variables. We studied robust and non-robust instances of Dekker and Peterson for two threads, as well as Lamport's fast mutex [22] for three threads. Moreover, we checked the CLH and MCS locks: robust list-based queue locks that use compare-and-set [16].

As second class of examples, we considered concurrent data structures. The Lock-Free Stack is a concurrent stack implementation using compare-and-swap [16]. Cilk's THE WSQ is a work stealing queue from the implementation of the Cilk-5 programming language [13].

Finally, we considered miscellaneous concurrent algorithms that are known to be sensitive to program order relaxations. We analysed several instances of the Non-Blocking Write protocol [18]. NBWL is the spinlock + non-blocking write example considered by Owens in Section 8 of [25]. Finally, our tool discovered the known bug in Java's Parker implementation that is due to TSO relaxations [12].

8.2 Benchmarking

We executed Trencher on the examples, using a machine with Intel(R) Core(TM) i5 CPU M 560 @ 2.67GHz (4 cores) running GNU/Linux. Table 2 summarizes the results. The columns T, L, and I give the numbers of threads, labels, and instructions in the examples. RQ is the number of reachability queries raised by Trencher. Provided the example is robust, this number is equal to the number of attacks $(\mathsf{t_A}, \mathsf{stinst}, \mathsf{ldinst})$. NR1 is the number of verification queries that were answered negatively by Trencher itself, without running SPIN. Such queries correspond to attacks where stinst cannot be delayed past ldinst because of memory fences or locked instructions in between. NR2 and R are the numbers of queries that are answered negatively/positively by the external model checker. Hence, RQ = NR1 + NR2 + R. F is the number of fences inserted.

The column Spin gives the total CPU time taken by SPIN and Clang, the C compiler, to produce a verifier executable (pan). The column Ver provides the total CPU time taken by Trencher and the external verifier. Real is the wall-clock time in seconds of processing an example. All times are given in seconds.

Table 2. Benchmarking results. The test inputs are available online [1].

Program	T	L	I	RQ	NR1	NR2	R	F	Spin	Ver	Real
Peterson (non-robust)	2	14	18	23	2	12	9	2	7.7	0.5	2.9
Peterson (robust)	2	16	20	12	12	0	0	0	0.0	0.0	0.0
Dekker (non-robust)	2	24	30	95	12	28	55	4	31.7	2.1	14.2
Dekker (robust)	2	32	38	30	30	0	0	0	0.0	0.0	0.0
Lamport (non-robust)	3	33	36	36	9	15	12	6	14.4	6.0	5.9
Lamport (robust)	3	39	42	27	27	0	0	0	0.0	0.0	0.0
CLH Lock (robust)	7	62	58	54	48	6	0	0	4.9	0.2	1.6
MCS Lock (robust)	4	52	50	30	26	4	0	0	2.9	0.4	0.9
Lock-Free Stack (robust)	4	46	50	14	14	0	0	0	0.0	0.0	0.0
Cilk's THE WSQ (non-robust)	5	86	79	152	141	8	3	3	10.0	18.0	7.4
NBW2 (non-robust)	2	21	19	15	9	5	1	1	2.5	0.2	0.8
NBW3 (robust)	2	22	20	15	15	0	0	0	0.0	0.0	0.0
NBW4 (robust)	3	25	22	9	7	2	0	0	0.7	0.1	0.4
NBWL (robust)	4	45	45	30	26	4	0	0	2.7	0.2	0.7
Parker (non-robust)	2	9	8	2	0	1	1	1	0.5	0.0	0.3
Parker (robust)	2	10	9	2	2	0	0	0	0.0	0.0	0.0

8.3 Discussion

The analysis of robust algorithms is particularly fast. They typically only have a small number of attacks that have to be checked by a model checker. The robust versions of Dekker and Peterson do not have such attacks at all. In the CLH and MCS locks, their number is less than 20%.

In some examples (non-robust Dekker, CLH Lock, NBW2, NBW4), up to 94% of the CPU time was spent on generating verifiers. This leaves room for improvement by switching to a model checker without compilation phase. For some examples (LamNR, CLH Lock), the wall-clock time constitutes 1/3 to 1/4 of the CPU time (4-cores). This confirms good parallelizability of the approach.

Remarkably, our trace-based analysis can establish robustness of the NBWL example, as opposed to the earlier analysis via triangular data races which has to place a fence [25].

We note that there is a reduction of TSO robustness to a *single* SC reachability query, again in an instrumented program of linear size. The idea is to let each thread act as an attacker and as a helper, and to apply $[\![-]\!]_{\mathsf{A1}}$ to all loads and stores rather than to a single attack. This alternative reduction is implemented in Trencher, but it performed worse in our experiments because of a higher degree of non-determinism and the lack of parallelization options.

Acknowledgements. The second 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.

References

1. Trencher: a tool for checking and enforcing robustness against TSO, http://concurrency.cs.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. Adve, S.V., Hill, M.D.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. 4(6), 613–624 (1993)
4. Alglave, J.: A Shared Memory Poetics. PhD thesis, University Paris 7 (2010)
5. 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)
6. Alur, R., McMillan, K., Peled, D.: Model-Checking of Correctness Conditions for Concurrent Objects. In: LICS, pp. 219–228. IEEE Computer Society Press (1996)
7. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the Verification Problem for Weak Memory Models. In: POPL, pp. 7–18. ACM (2010)
8. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. CoRR, abs/1208.6152 (2012), http://arxiv.org/abs/1208.6152
9. 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)
10. 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)
11. 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)
12. Dice, D.: A race in locksupport park() arising from weak memory models (November 2009), https://blogs.oracle.com/dave/entry/a_race_in_locksupport_park
13. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. SIGPLAN Not. 33(5), 212–223 (1998)
14. German, S.M., Sistla, P.A.: Reasoning about systems with many processes. JACM 39, 675–735 (1992)
15. Gibbons, P.B., Korach, E.: Testing shared memories. SIAM J. Comp. 26(4), 1208–1244 (1997)
16. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. MKP (2008)
17. Holzmann, G.J.: The model checker SPIN. IEEE Tr. Soft. Eng. 23, 279–295 (1997)
18. Kopetz, H., Reisinger, J.: The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronisation Problem. In: IEEE Real-Time Systems Symposium, pp. 131–137. IEEE Computer Society Press (1993)
19. Kuperstein, M., Vechev, M.T., Yahav, E.: Partial-Coherence Abstractions for Relaxed Memory Models. In: PLDI, pp. 187–198. ACM (2011)
20. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. SIGACT News 43(2), 108–123 (2012)
21. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comp. 28(9), 690–691 (1979)
22. Lamport, L.: A fast mutual exclusion algorithm. ACM Tr. Comp. Sys. 5(1) (1987)

23. Linden, A., Wolper, P.: A Verification-Based Approach to Memory Fence Insertion in Relaxed Memory Systems. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 144–160. Springer, Heidelberg (2011)
24. Lipton, R.: The reachability problem requires exponential space. Technical Report 62, Yale University (1976)
25. Owens, S.: Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010)
26. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO (extended version). Technical Report CL-TR-745, University of Cambridge (2009)
27. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comp. Sci. 6, 223–231 (1978)
28. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. CACM 53, 89–97 (2010)
29. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM TOPLAS 10(2), 282–312 (1988)
30. Vafeiadis, V., Zappa Nardelli, F.: Verifying Fence Elimination Optimisations. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 146–162. Springer, Heidelberg (2011)