The Best of Both Worlds: Trading Efficiency and Optimality in Fence Insertion for TSO*

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo

Department of Information and Technology, Uppsala University, Sweden
{parosh,mohamed_faouzi.atig,tuan-phong.ngo}@it.uu.se

Abstract. We present a method for automatic fence insertion in concurrent programs running under weak memory models that provides the best known trade-off between efficiency and optimality. On the one hand, the method can efficiently handle complex aspects of program behaviors such as unbounded buffers and large numbers of processes. On the other hand, it is able to find small sets of fences needed for ensuring correctness of the program. To this end, we propose a novel notion of correctness, called *persistence*, that compares the behavior of the program under the weak memory semantics with that under the classical interleaving (SC) semantics. We instantiate our framework for the Total Store Ordering (TSO) memory model, and give an algorithm that reduces the fence insertion problem under TSO to the reachability problem for programs running under SC. Furthermore, we provide an abstraction scheme that substantially increases scalability to large numbers of processes. Based on our method, we have implemented a tool and run it successfully on a wide range benchmarks.

1 Introduction

Most modern processor architectures implement weak (relaxed) memory models for performance reasons [2,15]. However, this comes at a price since a program may exhibit behaviors that deviate substantially from its behaviors under the usual *Sequentially Consistent (SC)* semantics. The standard way to eliminate the undesired behaviors is to insert memory *fence* instructions that typically prevent reordering of instructions issued before and after the fence. The most common model corresponds to TSO (for Total Store Ordering) that is adopted by Sun's SPARC multiprocessors and x86 multiprocessors [25,26].

Challenge. An important problem in concurrent programming is to find sets of fences that ensure program correctness without compromising efficiency. Manual fence placement is time-consuming and error-prone due to complex behaviors introduced by weak memory models. The challenge then is to develop methods

* An open source tool with all the experimental data are publicly available at https://github.com/PhongNgo/persistence

© Springer-Verlag Berlin Heidelberg 2015
J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 308–332, 2015.
DOI: 10.1007/978-3-662-46669-8_13

for *automatic* fence placement. A fence insertion algorithm requires an underlying verification algorithm that checks the correctness of the program for a given set of fences. This is necessary in order to be able to decide whether the current set of fences is sufficient, or whether additional fences are needed to achieve correctness. Designing such an algorithm is hard since we face a crucial trade-off between two criteria, namely:

• *Efficiency.* The algorithm needs to be able to carry out efficient analysis of complex program behaviors that arise due to intricate reorderings of program events. This complexity is for instance reflected by the fact that standard operational definitions for weak memory models [25,26] use *unbounded store-buffer* semantics, thus giving rise to an infinite state space even in the case where the original program is finite-state. Furthermore, since we are dealing with concurrent programs, the algorithm should scale well when increasing the number of processes and the number of variables.

• *Optimality.* The algorithm should derive sets of fences that are as close to optimal as possible. More precisely, we are required to avoid *under-fencing*, i.e., inserting too few fences since this would result in unsound program behaviors; and avoid *over-fencing*, i.e., inserting too many fences since this would result in a degradation of program performance (see e.g., [3,14,9,13], for descriptions of the high cost of fences on CPU-intensive concurrent programs).

In this context, identifying "good correctness properties" is crucial since a given property represents a particular choice in the trade-off between efficiency and optimality. For instance, at one extreme, we may require that the program is *data race free* (DRF) under SC (e.g. [24,22]). However, this will cause over-fencing, and hence failing the optimality criterion (see §2). In fact, some data races are in reality not harmful. For example, two *racy* implementations of a work-stealing queue [23,19] perform well under TSO without requiring fences. At the other extreme, we may consider *SC properties* such as safety and liveness properties. This would result in smaller sets of fences than in the previous case, but the verification problem becomes significantly harder (a *non-primitive recursive* lower-bound) or even undecidable [6,7], thus failing the efficiency criterion. Between these two extremes, the works in [11,5,9] consider the *robustness* (called also stability) property, i.e., checking whether a program generates the same set of traces *à la* Shasa and Snir [27] under weak memory and SC semantics. Robustness represents a correctness criterion between DRF property and SC properties since it is a weaker condition than the former and hence it would generate smaller sets of fences, while it could be more efficient than the latter since its verification problem belongs to a lower complexity class for finite-state programs under TSO (PSPACE-COMPLETE [10]). Robustness can be checked through a reduction to the reachability problem for a set of target programs under SC [9]. However, checking robustness causes state space explosion (see an explanation in the related work, §2), and furthermore, robustness may insert unnecessary fences as demonstrated by our experimental results.

Contribution. In this paper, we present a tool for automatic fence insertion in concurrent programs running under TSO that gives a good trade-off between efficiency and optimality. To this end, we make the following contributions.

• *Persistence:* We introduce a novel notion of correctness, called *persistence*, that as demonstrated by our experimental data provides a good trade-off between efficiency and optimality. Persistence considers the traces of a program and extracts two parameters, namely (i) *program order*: the order in which instructions are executed within the same process; and (ii) *store order*: the order in which different write operations hit the shared memory. The program is deemed to be persistent if it generates the same program and store orders under the TSO and SC semantics. If a program is persistent then it reaches identical sets of configurations (a configuration is a global state of the program) under TSO and SC. In particular, if the program is correct wrt. a given safety property under SC, then it will also be correct wrt. the same property under TSO.

• *Pattern:* We present an algorithm that automatically reduces the problem of checking persistence to the problem checking whether a given program exhibits a certain behavior pattern under SC . Despite the high complexity of the proof, the definition of the pattern is extremely simple. Crucially, from the efficiency point of view: (i) we need only to perform one reachability analysis query on a single target program, and (ii) there is no explosion in the size of the target program, since it contains the same number of processes, and only two extra variables compared to the original program (regardless of the number of processes).

• *Fence Insertion:* We present an algorithm that produces a *minimal* set of fences needed to ensure the program is persistent. The set is minimal in the sense that removing any fence in the set makes the program non-persistent. The algorithm is *counter-example-guided*, using counter-examples that are produced by the persistence checking algorithm. The fact that we reduce checking persistence to reachability analysis of SC programs allows using existing tools for program verification (we use Spin [16] in the current implementation of our tool).

• *Abstraction:* We present a general abstraction framework that is compatible with the notion of persistence, in the sense that persistence of the abstract program implies persistence of the concrete program. We instantiate the framework by defining an abstraction function that allows to reduce the number of variables, thus significantly limiting the state space explosion problem.

• *Tool:* We have implemented an open-source and publicly available tool, called Persist, that we use to evaluate our framework on a wide range of benchmarks. Persist uses Spin as a backend tool for checking reachability queries for programs under SC. Since Spin runs on finite-state programs, our experiments are carried out only on such programs. We do an extensive comparison with state-of-the-art tools, such as Trencher [9], Memorax [1], Remmex [20] , and Musketeer [3]. Our data shows that persistence indeed provides a good trade-off between efficiency and optimality.

2 Related Work

Fig. 1 shows the relevant correctness criteria ordered according to their strength. In this paper, we consider the *Persistence* condition (Per). The strongest condition is Data Race Freedom (Drf) [24,22] where the program is declared incorrect in case it contains a trace with a data race. The main drawback of this approach is over-fencing. In view of this, more precise techniques, based on weaker conditions, have been developed to uncover real violations.

In [24], *triangular race freedom* (Trf) is introduced where a program is considered to be correct if the traces of the program under TSO and SC agree on (i) program order, (ii) store order, and (iii) the source relation (in some works, called the read-from relation). Condition (iii) records the write operation from which a given read operation fetches its value. The main limitation of Trf approach is that it does not come with a method for checking program correctness w.r.t. Trf. Our approach is a weakening of Trf in the sense that we have removed the source relation, and therefore using Trf will cause over-fencing compared to our method. Observe that the pattern for checking persistence (despite the high complexity of the proof) is similar to the pattern for checking Trf. Hence, we can remove the read-from relation from TRF without paying a huge cost.

Another weakening of the Trf condition is *Robustness* (Rob) (known also as stability) [27,11,12,5,9], where the store order condition is replaced by a weaker condition, namely *variable store order* (sometimes called *coherence* order). The latter considers the order of memory updates performed on each variable individually. In [9], a tool (called Trencher) is provided for *exact* checking of the robustness criterion. Our approach offers two advantages over this approach: (i) *Efficiency*: In [9], the robustness problem for TSO is reduced to the reachability problem for a set of target programs under the SC semantics. However, the number of reachability queries issued, i.e., the size of the set of target programs, is quadratic in the size of the original program (this number is given by the number of pairs of instructions that can be reordered). Furthermore, the reduction triples the number of variables in each target program compared to the original program. In contrast, we reduce the persistence problem to a single reachability query for a program under SC which contains only 2 additional variables compared the original program (regardless of its number of variables or processes). This means that checking robustness is much more sensitive to the state space explosion problem than checking persistence (which is also visible in our experimental results, where we use the same backend tool Spin). (ii) *Optimality*: Although the approaches are incomparable in general from the point of view of optimality (robustness-based analysis may insert fewer or more fences than persistence-based analysis), the absence of the source relation implies, in almost all examples, that we insert at most the same number of fences. In fact,

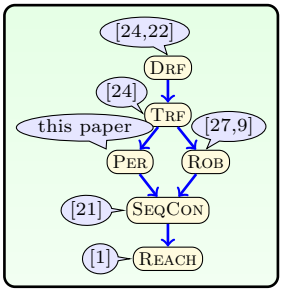

Fig. 1. Correctness criteria

in several examples, we insert a strict subset of the set of fences inserted by Trencher. Finally, no abstraction techniques are known for checking robustness.

Tools have been developed for *approximate* analysis of robustness (e.g., [11,12,5,3]). For instance, Musketeer [3] is based on static detection of critical cycles (that may violate robustness) in the control-flow graph of the program. The tool scales well to large programs but may cause over-fencing. In all examples we consider, we insert a subset of the set of fences inserted by Musketeer.

Liu et al. [21] consider even weaker conditions. One of them is *Sequential Consistence* (SeqCon) (called *state-based robustness* in [9]), i.e., checking whether there are any states of the program that are reachable under the weak memory semantics, but not under SC. The weakest condition is considered in [1], where the method checks Reach, i.e., whether a given state of the program is reachable under the weak memory semantics. The latter approach is used to implement Memorax which is a sound and complete tool for correcting finite-state programs under TSO wrt. safety properties. In contrast to our approach, checking the conditions SeqCon and Reach have non-primitive recursive complexities even for finite-state programs (as shown in [6,1]). This is reflected in our experimentation by the number of cases in which Memorax runs out of time/memory.

Several approximate tools have been developed for checking SC properties for programs (e.g., [21,8,17,20,4,18]). For instance, Remmex [20] is a state-space exploration tool with acceleration techniques. Remmex suffers from the state-space explosion problem as shown by our experimental data (see §11).

3 Overview

We will give an overview of the main ingredients of our framework, illustrating the definitions and algorithms through a simple toy example. We present our model for describing concurrent programs, and then introduce the notions of *runs* and *traces* using them to define the notion of *persistence*. A non-persistent program is said to be *fragile*. We solve the problem of checking whether a given program is persistent (or fragile) in two steps. First, we show that any fragile program has a run containing a certain *fragility pattern*. Second, we reduce the problem of checking the existence of runs with fragility patterns in a given program $\mathcal{P}$ to the *reachability problem* under SC for a target programs that we derive automatically from $\mathcal{P}$. Then, we present our counter-example guided fence insertion procedure. Finally, we introduce our abstraction technique.

```
program simple   vars x,y,z,t   procs
process p1                    process p2
regs $r1  init q1             regs $r2 init s1
begin                         begin
 q1: t=1; goto q2              s1: t=1; goto s2
 q2: x=2; goto q2              s2: y=1; goto s3
 q2: z=1; goto q3              s3: $r2=t; goto s4
 q3: $r1=x; goto q4            s4: x= $r2; goto s5
end                           end
```

Fig. 2. A simple running example

Model. Fig. 2 shows an example of a toy program, named *simple*, consisting of two concurrent processes (threads), called p_1 and p_2. Communication

between processes is performed through four shared variables t, x, y, and z to which the processes can read and write. The processes have one register each, namely $\$r_1$ and $\$r_2$. The registers and shared variables are allowed to range over infinite domains (here they range over set of integers). The behavior of a process is defined by a list of assembly-like instructions, each consisting of a label and a statement such as a read or write statement (see §5 for the full list of statements). For instance, at q_1, process p_1 performs a write statement in which it assigns the value 1 to the shared variable t. Notice that there are two instructions in p_1 labeled with q_2. This means that, once p_1 has executed the instruction labeled with q_1, it may non-deterministically choose to move to any of the two instructions labeled with q_2. This allows to encode conditional branching, iteration, and non-determinism, all using the same light syntax.

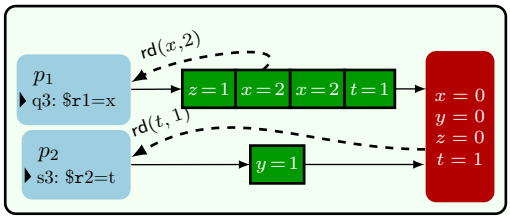

Fig. 3. Store buffers and the shared memory of a program under TSO

To define the runs of the program, we will use the operational semantics for TSO given in [25,26]. Conceptually, the model adds a FIFO buffer, called a *store buffer*, between each process and the main memory (cf. Fig. 3). The buffer is used to store the write operations performed by the process. A process executing a write instruction inserts it into its store buffer and immediately continues executing subsequent instructions. Memory updates are performed by non-deterministically choosing a process and by executing the oldest write operation in its buffer (the right-most element in the buffer). If a process p performs a read operation on a variable x then there are two possible scenarios. More precisely, if the buffer contains some write operations on x, then the read value must correspond to the value of the most recent such a write operation (the one that lies closest to the entry of the buffer). Otherwise, the value is fetched from the memory. Essentially, this means that a read operation on a variable x may *overtake* a sequence of write operations stored in its own buffer provided that all these operations concern variables that are different from x. For example, in the given configuration of Fig. 3, p_1 can read the value 2 from x, and p_2 can read the value 1 from t. A fence means that the buffer of the process must be flushed before the program can continue beyond the fence. The store buffers of processes are *unbounded* since there is *a priori* no limit on the number of write operations that can be issued by a process before a

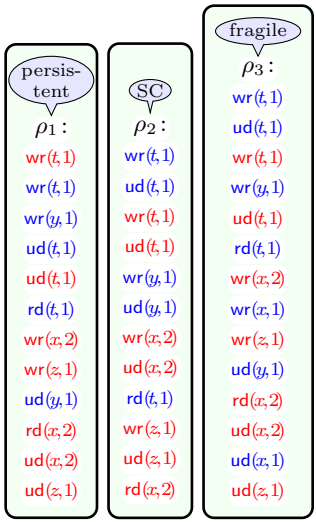

Fig. 4. Runs from configuration d of Fig. 9

memory update occurs. For instance, in the program of Fig. 2 the loop labeled by q_2 in process p_1 may generate an unbounded number of write operations, and hence create an unbounded number of elements in the buffer of p_1.

Runs. Consider the program of Fig. 2. In Fig. 4 we depict three typical runs ρ_1, ρ_2, ρ_3 of the program. A run consists of a sequence of *events*. We define the notion of an event formally in §5. In this section, it is sufficient to think of an event as an instruction executed by the process. The runs start from the initial configuration d of the program (depicted in Fig. 9). A configuration represents the (global) state of the program. In the configuration d, the processes are about to execute the instructions labeled by q_1 and s_1, and the values of all the shared variables and registers are equal to 0, while the store buffers are empty.

We describe ρ_1 below. To simplify the presentation, we identify an event with the operation that the process performs. For instance, we write the first step of process p_1 in ρ_1 as $\mathsf{wr}(t,1)$ since p_1 will perform a write operation in which it assigns 1 to x. First, three write events are issued by the processes, namely $\mathsf{wr}(t,1)$ by p_1, and $\mathsf{wr}(t,1)$, $\mathsf{wr}(y,1)$ by p_2. The new values are stored inside the corresponding buffers. Next, the two update events $\mathsf{ud}(t,1)$ and $\mathsf{ud}(t,1)$ are performed by the processes after which the value of t in the memory will be equal to 1, and the read event $\mathsf{rd}(t,1)$ can be performed by p_2, where $\$\mathtt{r}_2$ will be assigned the value 1. After the next three events $\mathsf{wr}(x,2), \mathsf{wr}(z,1), \mathsf{ud}(y,1)$, the buffer of p_1 contains two write events $\mathsf{wr}(x,2)$ and $\mathsf{wr}(z,1)$, which means that p_1 can read the value 2 for x from the buffer. Finally, the last two update events will be performed and both buffers will now be empty.

Traces. The *trace* of a run π, records (i) the *program order*, i.e., the order of the read and write events executed by each process in π. The program order of ρ_1 in Fig. 4 is given by $\mathsf{wr}(t,1)\mathsf{wr}(x,2)\mathsf{wr}(z,1)\mathsf{rd}(x,2)$ for p_1 and by $\mathsf{wr}(t,1)\mathsf{wr}(y,1)\mathsf{rd}(t,1)$ for p_2. (ii) the *store order* is the sequence of memory updates performed during the run. This is equal to $\mathsf{ud}(t,1)\mathsf{ud}(t,1)\mathsf{ud}(y,1)\mathsf{ud}(x,2)\mathsf{ud}(z,1)$ for ρ_1. The trace of ρ_1 is depicted in Fig. 5. Arrows labeled by *po* indicate the program order, e.g., the arrow from $\mathsf{wr}(y,1)$ to $\mathsf{rd}(t,1)$. Arrows labeled by *so* indicate the store order. For instance, the arrow from $\mathsf{wr}(y,1)$ to $\mathsf{wr}(x,2)$ means that the event $\mathsf{ud}(y,1)$ corresponding to $\mathsf{wr}(y,1)$ occurs before the event $\mathsf{ud}(x,2)$ corresponding to $\mathsf{wr}(x,2)$.

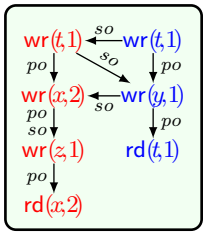

Fig. 5. The trace of ρ_1 and ρ_2 in Fig. 4

Persistence. A run is *Sequentially Consistent (SC)* if every write event is immediately followed by the corresponding update. Intuitively, this means that write events are atomic. An example is the run ρ_2 in Fig. 4. A run π is *persistent* if there is an SC run π' such that the traces of π and π' are identical, otherwise we say that π is *fragile*. For instance, in Fig. 4, the run ρ_1 is persistent since its trace is identical to that of ρ_2 (namely the trace shown in Fig. 5). We argue that the run ρ_3 in Fig. 4 is fragile. If not, there is an SC run ρ, with a trace identical

to the one in Fig. 6. From the equality of store orders of ρ and ρ_3 and the fact that the update events, $\mathsf{ud}(x,2)$, $\mathsf{ud}(x,1)$, and $\mathsf{ud}(z,1)$, occur in that order in ρ_3, it follows that the corresponding write events, $\mathsf{wr}(x,2)$, $\mathsf{wr}(x,1)$, and $\mathsf{wr}(z,1)$, will occur in the same order in ρ. From the equality of program orders it follows that $\mathsf{rd}(x,2)$ will occur after $\mathsf{wr}(z,1)$ (and hence also after $\mathsf{wr}(x,1)$) in ρ. Then ρ contains the sequence $\mathsf{wr}(x,2)\mathsf{wr}(x,1)\mathsf{wr}(z,1)\mathsf{rd}(x,2)$ which is not possible.

A program is *persistent* if all runs from its initial configuration are persistent, otherwise it is fragile. In the *persistence problem*, we check whether a given program is persistent or fragile. Notice that a persistent program reaches the same set of configurations under TSO and SC, and hence it satisfies the same safety properties under SC and TSO (see §6, paragraph on Safety Properties).

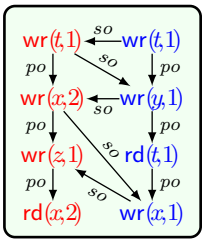

Fig. 6. The trace of ρ_3 in Fig. 4

Patterns. We reduce the persistence problem to the problem of checking the existence of a certain type of runs, called runs of type $\circledast$ (see Theorem 1). A run of of type $\circledast$ is defined with respect to one of the processes, called the *pivot* of the run (the other processes are called *fringe* processes). Fig. 7 shows an example of such a run for the program of Fig. 2, ρ_*, where the pivot is p_1 and the (only) fringe process is p_2. A run π of type $\circledast$ is the concatenation $\pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4$ of four parts. In addition to being SC, π satisfies a number of conditions. We do not place any constraints on the first part π_1 (except that is it SC). The second part π_2, consists of two events performed by the pivot, namely a write event e_1 followed by the matching update event u_1. In our example, p_1 writes the value 1 to the variable z in π_2. The third part π_3 (which is empty in our example) consists only of events performed by the pivot, although it is not allowed to perform any write, update, atomic-read-write, or fence events. The fourth part π_4, consists of two events performed by the fringe process, namely a write event e_2 followed by the matching update event u_2. Furthermore, the variable updated here should be different from the variable updated in π_2. In our example, p_2 writes the value 1 to the variable x (which is different from z). Finally, there should be a read event e of the pivot (the event corresponding to the instruction labeled q_3 in our example), called the *complementary event* of π, such that the following properties hold: (i) e should be enabled after $\pi_1 \cdot \pi_2 \cdot \pi_3$. In our example, the event labeled q_3 is enabled after $\pi_1 \cdot \mathsf{wr}(z,1)\mathsf{ud}(z,1)$. (ii) In e, the process reads a value from the same variable as the one updated in π_4. In our example, this variable is x. (iii) The value read during e (i.e., the value of the read variable in the memory after $\pi_1 \cdot \pi_2 \cdot \pi_3$) should be different from the value assigned to it in π_4. In our example, the value of x in the memory after $\pi_1 \cdot \mathsf{wr}(z,1)\mathsf{ud}(z,1)$ is equal to 2 (which is different form the value 1 assigned to x in π_4).

The proof of existence of runs of type $\circledast$ is highly non-trivial. However, once done, it allows to define a surprisingly simple pattern for detecting fragile runs. More precisely, we can derive the fragile run $\pi_1 \cdot e_1 \cdot \pi_3 \cdot e \cdot e_2 \cdot u_2 \cdot u_1$, which we call the *witness*. In the above example, the witness is given by $\pi_1 \cdot \mathsf{wr}(z,1)\mathsf{rd}(x,2)\mathsf{wr}(x,1)\mathsf{ud}(x,1)\mathsf{ud}(z,1)$.

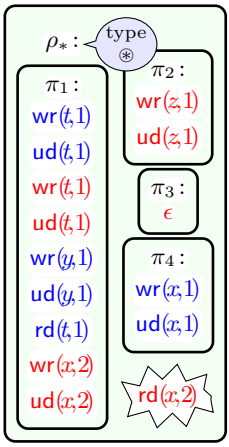

Fig. 7. Run of type $\circledast$ from d of Fig. 9

Pattern Detection. Given a program $\mathcal{P}$ we generate a new *target* program $\mathcal{Q}$ such that $\mathcal{P}$ contains a witness iff $\mathcal{Q}$ can reach a given set of states under SC. The program $\mathcal{Q}$ will run in three phases 0, 1, 2, and find the existence of a witness (as described above). Recall that such a witness is derived from a type $\circledast$ run $\pi = \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4$ which is defined wrt. pivot. In phase 0 (see Fig. 10), $\mathcal{Q}$ simulates π_1, and hence all the processes will simulate their moves in $\mathcal{P}$. At the end of phase 0 (see Fig. 11), one of the processes will non-deterministically decide to play the role of the pivot. At the same time, it records the variable on which it performs a write event in π_2 (in the above example, this variable is z). In our construction we do this by assigning a special value $\mathfrak{c}_1$ to the variable z. In phase 1 (see Fig. 12), $\mathcal{Q}$ simulates π_3 in which only the pivot is active. At the end of phase 1 (see Fig. 13), the pivot ensures the existence of the complementary event. This is done by (i) choosing an enabled read event, (ii) ensuring that the involved variable is different from the one that was updated during π_2 (at the end of phase 0). This can be done by checking that its value is different from $\mathfrak{c}_1$. (iii) It records the read event of the complementary event by copying the value of the variable x to a new variable new and announcing the variable x to the fringe processes by assigning to it a new value $\mathfrak{c}_2$. Finally, in phase 2 (see Fig. 14), a fringe process verifies the existence of π_4, by finding the (only) variable whose value is $\mathfrak{c}_2$ and check that it can indeed assign to it a different value from the one that was assigned by the complementary event (by comparing with the value stored in new).

An important aspect of our scheme is that $\mathcal{Q}$ contains only two additional variables compared to $\mathcal{P}$, namely the variable new, and a variable with a small domain that we use to record the current simulation phase. This holds regardless of the number of variables in $\mathcal{P}$. Thus, the verification problem for the target program is as efficient as that for the source program (when run under SC).

Fence Insertion. A naive way to find the minimal set of fences is to simply try out all combinations. Obviously, such an algorithm would not work in practice due to the large number of possible combinations. Instead, we use counter-examples analysis, where we use witnesses provided by our persistence detection algorithm. A witness of the form shown above is fragile since complementary event overtakes the event e_1. Therefore, inserting a fence somewhere along $e_1 \cdot \pi_3 \cdot e_2 \cdot u_2 \cdot u_1$ is both necessary and sufficient to disable the witness. This allows to derive a set of fences by repeatedly calling the pattern detection algorithm, each time inserting a new fence, until the program becomes persistent. Notice that when the program becomes persistent the pattern detection algorithm declares that no witnesses exists any more. The derived set is optimal since each of the inserted is necessary to eliminate a witness, and hence removing any of them would make the program fragile. In the program of Fig. 2, our algorithm would put a fence after the second instruction labeled by q_2 in p_1, making the program persistent.

Abstraction. We develop an abstraction framework, called *observation abstraction*, that exploits the fact that persistence is not sensitive to the *source* relation. A process needs only to know the value of a variable in its own buffer or in the memory. It does not need to figure out the process (or instruction) that produced this value. The idea is to reason about the memory view of each process p: Each process can only observe the memory changes due to its own instructions on shared variables or the changes caused by the other processes over the set variables that p can read from. We abstract the rest of the processes, in a way that p has at least the same sequences of memory views as in the original program $\mathcal{P}$. This ensures that the persistence of the abstract program implies persistence of $\mathcal{P}$. We instantiate our framework by defining an abstraction function, called *Flattening* (see Fig. 15), that can be used for efficient checking of persistence.

4 Preliminaries

We use $\mathbb{N}$ to denote the set of natural numbers. For sets A and B, we use $f : A \mapsto B$ to denote that f is a function that maps A to B. For $a \in A$ and $b \in B$, we use $f\,[a \hookleftarrow b]$ to denote the function f' where $f'(a) = b$ and $f'(a') = f(a')$ for all $a' \neq a$. Let A be a finite set. We use $|A|$ to denote its size. We use A^* (resp. A^+) to denote the set of words (resp. non-empty words) over A; and ϵ to denote the empty word. Consider a word $w = a_1 a_2 \cdots a_n \in A^*$. We define $|w| := n$ and $\mathsf{last}\,(w) := a_n$. For $i : 1 \leq i \leq n$, we define $w[i] := a_i$. For $a \in A$, we write $a \in w$ if a appears in w, i.e., $a = w[i]$ for some $i : 1 \leq i \leq |w|$. For $B \subseteq A$, we define $w \odot B := a_{i_1} a_{i_2} \cdots a_{i_m}$ to be the maximal subword of w such that $a_{i_j} \in B$ for all $j : 1 \leq j \leq m$, i.e., we keep the elements that belong to B; and define $w \otimes B := i_1 i_2 \cdots i_m$, i.e., it gives the sequence of indices of the elements that belong to B.

5 Concurrent Programs

We define the syntax and the semantics we use for concurrent processes communicating through shared memory. Moreover, we define SC computations as a subclass of TSO ones. Finally, we introduce the reachability problem.

```
⟨prog⟩ ::= program ⟨progid⟩
           vars ⟨var⟩* procs ⟨proc⟩*
⟨proc⟩ ::= process ⟨procid⟩ regs ⟨reg⟩*
           init ⟨label⟩ begin ⟨inst⟩* end
⟨inst⟩ ::= ⟨label⟩ : ⟨stmt⟩ ; goto ⟨label⟩
⟨stmt⟩ ::= ⟨var⟩ = ⟨exp⟩ | ⟨reg⟩ = ⟨var⟩
         | ⟨reg⟩ = ⟨exp⟩ | fence
         | arw( ⟨var⟩ , ⟨exp⟩ , ⟨exp⟩ )
         | skip | assume ⟨exp⟩
⟨exp⟩ ::= ⟨fun⟩ ( ⟨reg⟩* )
```

Fig. 8. Syntax of a concurrent program

Syntax. The syntax of a program is given in Fig. 8. In the following, we assume a program with name $\mathcal{P}$, a set of shared variables $\mathbb{X}$, and a set of processes $\mathsf{ProcSet}$. Each process $p \in \mathsf{ProcSet}$ has a list of registers $\mathbb{R}_p$ and a list of assembly-like instructions $\mathbb{I}_p$. Each process starts from a statement with initial label $init_p$. We let $\mathbb{Q}_p$ be the set of labels that occur in the instructions of

process p, and assume w.l.o.g. that the sets of labels and also the sets of registers of the different processes are disjoint. We define the sets $\mathbb{R} := \cup_{p \in \mathsf{ProcSet}} \mathbb{R}_p$, $\mathbb{I} := \cup_{p \in \mathsf{ProcSet}} \mathbb{I}_p$, and $\mathbb{Q} := \cup_{p \in \mathsf{ProcSet}} \mathbb{Q}_p$. Sometimes, we represent a program $\mathcal{P}$ by a tuple $\langle \mathbb{X}, \mathsf{ProcSet}, \mathbb{Q}, \mathbb{I}, \mathit{init} \rangle$, where $\mathbb{X}$ is the set of instructions, $\mathsf{ProcSet}$ is the set of processes, $\mathbb{Q}$ is the set of labels, $\mathbb{I}$ is the set of instructions, and $\mathit{init} : \mathsf{ProcSet} \mapsto \mathbb{Q}$ is mapping such that $\mathit{init}(p) = \mathit{init}_p$ for all $p \in \mathsf{ProcSet}$.

The registers and shared variables range over some (potentially) infinite domain V. Here, we assume w.l.o.g. that V is the set of integers. An instruction $\mathtt{ins}$ is of the form $q_1 : \mathfrak{s}; \mathtt{goto}\ q_2$, where q_1, q_2 are labels, and $\mathfrak{s}$ is a *statement*. After the program has executed the statement $\mathfrak{s}$ it jumps to an instruction labeled with q_2. If several instructions are labeled with q_2, the process chooses one of them non-deterministically. If none exists, the process terminates. We assume that a program comes with a set $\mathbb{F}$ of functions. Our method is not dependent on the particular set of functions that occur in the programs, and therefore we will not specify the set precisely in the grammar. The set may include all the standard functions, such as addition, subtraction, multiplication, division, etc. An *expression* $\mathfrak{e}$ is either a constant (a member of V), register $\$\mathtt{r}$, or of the form $f(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)$ where $f \in \mathbb{F}$ is a function and $\mathfrak{e}_1, \ldots, \mathfrak{e}_n$ are expressions. A statement $\mathfrak{s}$ is one of the following forms: (i) wr (write statement): $x = \mathfrak{e}$ writes the value of the expression $\mathfrak{e}$ to the shared variable x. This value will be stored in the buffer of the process. (ii) rd (read statement): $\$\mathtt{r} = x$ reads the value of the shared variable x (either from the buffer of the process or from the memory), and stores it in the register $\$\mathtt{r}$. (iii) arw ($\mathsf{atomic\text{-}read\text{-}write}$ statement): $\mathsf{arw}(x, \mathfrak{e}_1, \mathfrak{e}_2)$ checks atomically whether the value of the shared variable x is equal to the value of $\mathfrak{e}_1$; if true it assigns the value of $\mathfrak{e}_2$ to x, otherwise the execution of the instruction is blocked. (iv) fn (fence statement): flushes the buffer of the process. (v) skip statement: is the empty statement. (vi) asgn (assign statement): $\$\mathtt{r} = \mathfrak{e}$ assigns the value of the expression $\mathfrak{e}$ to the register $\$\mathtt{r}$. (vii) asm (assume statement): $\mathsf{assume}\,\mathfrak{e}$ checks whether $\mathfrak{e}$ evaluates to *true*. If not, the execution of the instruction is blocked. We use *source* ($\mathtt{ins}$), *stmt* ($\mathtt{ins}$), and *target* ($\mathtt{ins}$), to denote q_1, $\mathfrak{s}$, and q_2 respectively. We classify instructions according to the forms of their statements. First, for a process p, and $i \in \{\mathsf{wr}, \mathsf{rd}, \mathsf{arw}, \mathsf{fn}, \mathsf{skip}, \mathsf{asgn}, \mathsf{asm}\}$, we define $\mathbb{I}_p^i$ to be the set of instructions in $\mathbb{I}_p$ with an i statement. For instance, $\mathbb{I}_p^{\mathsf{wr}}$ consists of the instructions $\mathtt{ins} \in \mathbb{I}_p$ with write statements, i.e., *stmt* ($\mathtt{ins}$) is of the form $x = \mathfrak{e}$ for some x and $\mathfrak{e}$. Furthermore, for $i \in \{\mathsf{wr}, \mathsf{rd}, \mathsf{arw}\}$, and a variable $x \in \mathbb{X}$ we define $\mathbb{I}_p^{i,x}$ to be the set of instructions in $\mathbb{I}_p^i$ that operate on the variable x. For instance, $\mathbb{I}_p^{\mathsf{wr},x}$ consists of the instructions $\mathtt{ins} \in \mathbb{I}_p$ such that *stmt* ($\mathtt{ins}$) is of the form $x = \mathfrak{e}$ for some $\mathfrak{e}$. In the program of Fig. 2, the instruction labeled with q_1 is a member of $\mathbb{I}_{p_1}^{\mathsf{wr},t}$.

Configurations. To define configurations, we introduce the following concepts. A *label definition* $\overline{q} : \mathsf{ProcSet} \mapsto \mathbb{Q}$ is a function such that $\overline{q}(p) \in \mathbb{Q}_p$ for each $p \in \mathsf{ProcSet}$. Intuitively, for a process $p \in \mathsf{ProcSet}$, $\overline{q}(p)$ gives the label of the instruction that p will execute in its next step. A *register*

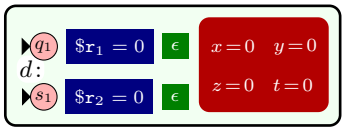

Fig. 9. Initial configuration d

state is a function $\overline{r} : \mathbb{R} \mapsto V$. For a register $\$\mathtt{r} \in \mathbb{R}$, the value of $\overline{r}(\$\mathtt{r})$ is the content of $\$\mathtt{r}$. A *buffer state* is a function $\overline{b} : \mathsf{ProcSet} \mapsto (\mathbb{X} \times V)^*$. The value of $\overline{b}(p)$ is the content of the buffer belonging to p. This buffer contains a sequence of write operations, where each write operation is defined by a pair, namely a variable x and a value v that is assigned to x. In our model, messages will be appended to the buffer from the left, and fetched from the right. A *memory state* is a function $mem : \mathbb{X} \mapsto V$ that defines the value of each variable in the memory. A *configuration* c is a tuple $\langle \overline{q}, \overline{r}, \overline{b}, mem \rangle$ where $\overline{q}$ is a label definition, $\overline{r}$ is a register state, $\overline{b}$ is a buffer state, and mem is a memory state. We use $\mathtt{LabelOf}\,(c)$, $\mathtt{BuffersOf}\,(c)$, $\mathtt{RegsOf}\,(c)$, and $\mathtt{MemoryOf}\,(c)$ to denote $\overline{q}$, $\overline{r}$, $\overline{b}$, and mem respectively. We define $\mathsf{EmptyBuffer}$ to be the buffer state such that $\mathsf{EmptyBuffer}(p) = \epsilon$ for all processes $p \in \mathsf{ProcSet}$. We say that c is *plain* if $\mathtt{BuffersOf}\,(c) = \mathsf{EmptyBuffer}$. The *initial configuration* c_{init} is defined by $\langle \overline{q}_{init}, \overline{r}_{init}, \mathsf{EmptyBuffer}, mem_{init} \rangle$ where $\overline{r}_{init}(\$\mathtt{r}) = 0$ for all $\$\mathtt{r} \in \mathbb{R}$, $\overline{q}_{init}(p) = init_p$ for all $p \in \mathsf{ProcSet}$, and $mem_{init}(x) = 0$ for all $x \in \mathbb{X}$. In other words, to start with, each process is in the initial label, all buffers are empty, and all registers and shared variables have value 0. We use C to denote the set of all configurations. For a configuration c and an expression $\mathfrak{e}$, we define the *evaluation* $c(\mathfrak{e})$ of $\mathfrak{e}$ in c inductively by $c(\$\mathtt{r}) := \mathtt{RegsOf}\,(c)\,(\$\mathtt{r})$, and $c(f(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)) := f(c(\mathfrak{e}_1), \ldots, c(\mathfrak{e}_n))$ (we regard a constant expression as a function with zero arguments). Fig. 9 illustrates an initial configuration d of the program in Fig. 2.

Events. We introduce the notion of *events* that describe two aspects of process behavior, namely the internal actions of a process and its interaction with the memory. The former consists of fence, skip, assign, and assume events, while the latter consists of write, read, and atomic-read-write events. For these three types of events we will also record the variable involved together with its value. Formally, we define the set of events Δ as follows. Let $p \in \mathsf{ProcSet}$ be a process. For $i \in \{\mathsf{fn}, \mathsf{skip}, \mathsf{asgn}, \mathsf{asm}\}$, we define the set $\Delta_p^i := \mathbb{I}_p^i$, i.e., for internal events, we define the set simply to be the set of instructions. For a variable $x \in \mathbb{X}$, a value $v \in V$, and $i \in \{\mathsf{wr}, \mathsf{rd}, \mathsf{arw}\}$, we define $\Delta_p^{i,x,v} := \mathbb{I}_p^{i,x} \times \{v\}$, i.e., the event is a pair including the instruction (whose statement operates on x) and the used value v. We define the sets $\Delta_p^{i,x} := \cup_{v \in V} \Delta_p^{i,x,v}$, $\Delta_p^i := \cup_{x \in \mathbb{X}} \Delta_p^{i,x}$, $\Delta_p := \cup_{i \in \{\mathsf{wr},\mathsf{rd},\mathsf{arw},\mathsf{fn},\mathsf{skip},\mathsf{asgn},\mathsf{asm}\}} \Delta_p^i$, and $\Delta := \cup_{p \in \mathsf{ProcSet}} \Delta_p$. For an event $e \in \Delta$ of the form $\mathfrak{ins}$ or the form $\langle \mathfrak{ins}, i \rangle$, we define $source\,(e) := source\,(\mathfrak{ins})$, $stmt\,(e) := stmt\,(\mathfrak{ins})$, and $target\,(e) := target\,(\mathfrak{ins})$. Together with the above sets of events (that are induced by the instructions of the processes), we define an additional type of events, namely update events as follows. For a process $p \in \mathsf{ProcSet}$, a variable $x \in \mathbb{X}$ and value $v \in V$, we define an event $\mathsf{ud}_p(x, v)$. We will use this event in the semantics to update the memory using the oldest message (x, v) in the buffer of process p. We define the sets $\Delta_p^{\mathsf{ud},x,v} := \{\mathsf{ud}_p(x, v)\}$, $\Delta_p^{\mathsf{ud},x} := \{\mathsf{ud}_p(x, v) \mid v \in V\}$, $\Delta_p^{\mathsf{ud}} := \cup_{x \in \mathbb{X}} \Delta_p^{\mathsf{ud},x}$, and $\Delta^{\mathsf{ud}} := \cup_{p \in \mathsf{ProcSet}} \Delta_p^{\mathsf{ud}}$. Furthermore, we define $\Delta_p^\bullet := \Delta_p \cup \Delta_p^{\mathsf{ud}}$ and $\Delta^\bullet := \Delta \cup \Delta^{\mathsf{ud}}$. In Fig. 2, let $\mathfrak{ins}_1$ (resp. $\mathfrak{ins}_2$) be the instruction labeled by q_1 (resp. s_3) in p_1 (resp. p_2). Then, $\langle \mathfrak{ins}_1, 1 \rangle \in \Delta_{p_1}^{\mathsf{wr},t,1}$, and $\langle \mathfrak{ins}_2, 1 \rangle \in \Delta_{p_2}^{\mathsf{rd},t,1}$. Furthermore, $\mathsf{ud}_{p_1}(x, 2) \in \Delta_{p_1}^{\mathsf{ud},x,2}$.

Transition Relation. We define the transition relation $\longrightarrow \subseteq C \times \Delta^\bullet \times C$ as follows. For configurations $c = \langle \overline{q}, \overline{r}, \overline{b}, mem \rangle$, $c' = \left\langle \overline{q}', \overline{r}', \overline{b}', mem' \right\rangle$, $p \in \mathsf{ProcSet}$, and $e \in \Delta_p^\bullet$, we write $c \xrightarrow{e} c'$ to denote that one of the following conditions holds:

- skip: $e \in \Delta_p^{\mathsf{skip}}$, $\overline{q}(p) = source\,(e)$, $\overline{q}' = \overline{q}\,[p \hookleftarrow target\,(e)]$, $\overline{r}' = \overline{r}$, $\overline{b}' = \overline{b}$, and $mem' = mem$. The process jumps to a statement with the target label, while the register, buffer, and memory contents remain unchanged.
- write: $e \in \Delta_p^{\mathsf{wr},x,v}$, $stmt\,(e)$ is of the form $x = \mathfrak{e}$ with $c(\mathfrak{e}) = v$, $\overline{q}(p) = source\,(e)$, $\overline{q}' = \overline{q}\,[p \hookleftarrow target\,(e)]$, $\overline{r}' = \overline{r}$, $\overline{b}' = \overline{b}\left[p \hookleftarrow (x,v) \cdot \overline{b}(p)\right]$, and $mem' = mem$.
- update: $e \in \Delta_p^{\mathsf{ud},x,v}$, $\overline{q}' = \overline{q}$, $\overline{r}' = \overline{r}$, $\overline{b} = \overline{b}'\left[p \hookleftarrow \overline{b}'(p) \cdot (x,v)\right]$, and $mem' = mem\,[x \hookleftarrow v]$.
- read: $e \in \Delta_p^{\mathsf{rd},x,v}$, $stmt\,(e)$ is of the form $\$\mathtt{r} = x$, $\overline{q}(p) = source\,(e)$, $\overline{q}' = \overline{q}\,[p \hookleftarrow target\,(e)]$, $\overline{r}' = \overline{r}\,[\$\mathtt{r} \hookleftarrow v]$, $\overline{b}' = \overline{b}$, $mem' = mem$, and one of the following conditions holds:
  - read-own-write: There is an $i : 1 \leq i \leq |\overline{b}(p)|$ s.t. $\overline{b}(p)[i] = (x,v)$, and there are no $1 \leq j < i$ and $v' \in V$ s.t. $\overline{b}(p)[j] = (x,v')$. If there is a write on x in the buffer of p then we consider the most recent of such a write (the left-most one in the buffer). This operation should assign v to x.
  - read-memory: $(x,v') \notin \overline{b}(p)$ for all $v' \in V$ and $mem(x) = v$. If there is no write operation on x in the buffer of p then the value v of x is fetched from the memory.
- fence: $e \in \Delta_p^{\mathsf{fn}}$, $\overline{q}(p) = source\,(e)$, $\overline{q}' = \overline{q}\,[p \hookleftarrow target\,(e)]$, $\overline{r}' = \overline{r}$, $\overline{b}(p) = \epsilon$, $\overline{b}' = \overline{b}$, and $mem' = mem$. A fence operation may be performed by a process only if its buffer is empty.
- arw: $e \in \Delta_p^{\mathsf{arw},x,v}$, $stmt\,(e)$ is of the form $\mathsf{arw}(x, \mathfrak{e}_1, \mathfrak{e}_2)$ with $c(\mathfrak{e}_2) = v$, $\overline{q}(p) = source\,(e)$, $\overline{q}' = \overline{q}\,[p \hookleftarrow target\,(e)]$, $\overline{r}' = \overline{r}$, $\overline{b}(p) = \epsilon$, $\overline{b}' = \overline{b}$, $mem(x) = c(\mathfrak{e}_1)$, and $mem' = mem\,[x \hookleftarrow v]$. The arw operation is performed by a process only if its buffer is empty. The operation checks whether the value of x is equal to the evaluation of $\mathfrak{e}_1$ in c. In such a case, it changes that value to v.
- assume: $e \in \Delta_p^{\mathsf{asm}}$, $stmt\,(e)$ is of the form $\mathsf{asm}\,\mathfrak{e}$ with $c(\mathfrak{e}) = true$, $\overline{q}(p) = source\,(e)$, $\overline{q}' = \overline{q}\,[p \hookleftarrow target\,(e)]$, $\overline{r}' = \overline{r}$, $\overline{b}' = \overline{b}$, and $mem' = mem$. The instruction can be performed only if $\mathfrak{e}$ evaluates to true in c.
- assign: $e \in \Delta_p^{\mathsf{asgn}}$, $stmt\,(e)$ is of the form $\$\mathtt{r} = \mathfrak{e}$, $\overline{q}(p) = source\,(e)$, $\overline{q}' = \overline{q}\,[p \hookleftarrow target\,(e)]$, $\overline{r}' = \overline{r}\,[\$\mathtt{r} \hookleftarrow c(\mathfrak{e})]$, $\overline{b}' = \overline{b}$, and $mem' = mem$. The content of $\$\mathtt{r}$ is updated to the value of $\mathfrak{e}$.

A event $e \in \Delta^\bullet$ is said to be *enabled* from a configuration c if there is a configuration c' with $c \xrightarrow{e} c'$. If e is enabled from c, then we use $e(c)$ to denote the unique c' such that $c \xrightarrow{e} c'$. We define $\longrightarrow := \cup_{e \in \Delta^\bullet} \xrightarrow{e}$, and use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$.

Runs. A *run* π in $\mathcal{P}$ is a sequence of events $e_1 e_2 \cdots e_n \in (\Delta^\bullet)^*$. We generalize the notion of enabledness to runs, and say that π is *enabled* from a configuration c

(or simply π is a run *from* c) if $c_0 \xrightarrow{e_1} c_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} c_n$ for some configurations $c_0, c_1, \ldots, c_n$ with $c_0 = c$. In such a case, we define $\pi(c) := c_n$ (notice that c_n is unique given the configuration c and the run π). We write $c \xrightarrow{\pi} c'$ to denote that $\pi(c) = c'$. Notice that, for configurations c and c', we have that $c \xrightarrow{*} c'$ iff $c \xrightarrow{\pi} c'$ for some run π. For a run π, we define the relation $\mathsf{match}(\pi) \subseteq \mathbb{N} \times \mathbb{N}$ such that $\mathsf{match}(\pi)(j,k)$ $(1 \leq j < k \leq |\pi|)$ holds if and only if there is a process $p \in \mathsf{ProcSet}$ and $\ell \in \mathbb{N}$ where $\left(\pi \otimes \Delta_p^{\mathsf{wr}}\right)[\ell] = j$, and $\left(\pi \otimes \Delta_p^{\mathsf{ud}}\right)[\ell] = k$. In other words e_j is the ℓ^{th} write operation performed by p, and e_k is the (matching) ℓ^{th} update operation performed by p. We use Π to denote the set of all runs, and use $\Pi(c)$ to denote the set of runs from c. For a set $\Pi' \subseteq \Pi$, we define $\Pi'(c) := \Pi(c) \cap \Pi'$, i.e., it is the subset of runs in Π' that are enabled from c. For a set of runs $\Pi' \subseteq \Pi$ and a set of events $\Delta' \subseteq \Delta$, we use $\Pi'(\Delta') := \Pi' \cap (\Delta')^*$, i.e., it is the subset of π' that uses only events from Δ'. For instance, $\Pi\left(\Delta_p^{\mathsf{rd}}\right)$ is the set of all runs consisting only of read events performed by p, $\Pi\left(\Delta_p^{\mathsf{rd}}\right)(c)$ is the subset of the latter enabled from c, and $\Pi\left(\neg\Delta_p^{\mathsf{rd}}\right)$ is the set of all runs that do not contain read events performed by p. We say that π is *complete* if $|\pi \odot \Delta^{\mathsf{wr}}| = |\pi \odot \Delta^{\mathsf{ud}}|$, i.e., the numbers of write and update events in π are equal. We use $\Pi^{\mathsf{Complete}}(c)$ to denote the set of runs from c that are complete. Notice that if c is plain and $\pi \in \Pi^{\mathsf{Complete}}(c)$ then $\pi(c)$ is plain.

Fig. 4 depicts different runs from the configuration d of Fig. 9 in the program *simple* of Fig. 2. All these runs are complete. To simplify the presentation of the runs in Fig. 4, we represent an event in $\Delta^{\mathsf{wr},x,v}$ by the triple $\mathsf{wr}(x,v)$, and an event in $\Delta^{\mathsf{rd},x,v}$ by the triple $\mathsf{rd}(x,v)$. This does not introduce any ambiguity, since each instruction in the program *simple* has a unique statement. For instance, the event $\mathsf{wr}(t,1)$ corresponds to the process p_1 performing the event labeled by q_1.

SC Semantics. We will define *SC* runs as special cases of *TSO* runs. For a process $p \in \mathsf{ProcSet}$, a run $\pi \in \Pi$ is said to be *Sequentially Consistent* (or *SC* for short) wrt. p if whenever $\pi[j] \in \Delta_p^{\mathsf{wr},x,v}$ then $1 \leq j < |\pi|$ and $\pi[j+1] \in \Delta_p^{\mathsf{ud},x,v}$. In other words, any write operation of the process p should be immediately followed by the matching update operation from the same process. We use Π_p^{SC} to denote the set of runs that are SC wrt. p. We say that π is *SC* if it is SC wrt. all processes $p \in \mathsf{ProcSet}$. We use Π^{SC} to denote the set of SC runs. We say that π is *singly TSO* wrt. $p \in \mathsf{ProcSet}$ if $\pi \in \Pi_r^{\mathsf{SC}}$ for all processes $r \in \mathsf{ProcSet} - \{p\}$, i.e., π is SC wrt. all processes except (possibly) p. We use $\Pi_p^{\mathsf{SinglyTSO}}$ to denote the set of runs that are singly TSO wrt. p. In Fig. 4, the run ρ_2 is SC, while ρ_1 is not singly TSO wrt. p_1 or p_2 (it is not SC wrt. p_1 or p_2).

Reachability Problem. An instance of the *reachability problem* is defined by a program and a finite set of label definitions Final. The question is whether there is a configuration c such that $c_{init} \xrightarrow{*} c$ for some configuration c with $\mathtt{LabelOf}(c) \in \mathsf{Final}$. Recall that $c_{init} \xrightarrow{*} c$ is equivalent to whether $c_{init} \xrightarrow{\pi} c$ for some run $\pi \in \Pi(c_{init})$. In the *SC reachability problem*, we restrict the program to SC runs, and ask whether there is a configuration c s.t. $c_{init} \xrightarrow{\pi} c$ for some c with $\mathtt{LabelOf}(c) \in \mathsf{Final}$ and $\pi \in \Pi^{\mathsf{SC}}(c_{init})$.

6 Persistence

We formulate the persistence problem by first introducing our notion of traces, and then comparing the set of traces of the program under TSO and SC. We explain the relation between persistence and correctness wrt. safety properties.

Traces. For a run π, we define the *program order* $\mathsf{ProgOrder}(\pi) : \mathsf{ProcSet} \mapsto \Delta^*$ by $\mathsf{ProgOrder}(\pi)(p) := \pi \odot \left(\Delta_p^{\mathsf{wr}} \cup \Delta_p^{\mathsf{rd}} \cup \Delta_p^{\mathsf{arw}}\right)$ for each $p \in \mathsf{ProcSet}$. In other words it extracts, for each process p, the sequence of write, read, and atomic-read-write events performed by the process. We define the *store order* by $\mathsf{StoreOrder}(\pi) := \pi \odot \left(\Delta^{\mathsf{ud}} \cup \Delta^{\mathsf{arw}}\right)$, i.e., it extracts the sequence of update and atomic-read-write events of all processes from π. Observe that, in the store order definition, we keep the two events that modify the memory. We define $\mathsf{Trace}(\pi)$ as $\langle \mathsf{ProgOrder}(\pi), \mathsf{StoreOrder}(\pi) \rangle$. Fig. 5 depicts a trace.

Persistence. Consider a plain configuration c and a complete run $\pi \in \Pi^{\mathsf{Complete}}(c)$. We say that π is *persistent* from c if there is an SC run $\pi' \in \Pi^{\mathsf{SC}}(c)$ with $\mathsf{Trace}(\pi) = \mathsf{Trace}(\pi')$; otherwise we say that π is *fragile* from c. We use $\Pi^{\mathsf{Persistent}}(c)$ and $\Pi^{\mathsf{Fragile}}(c)$ to denote the set of persistent and fragile runs from c respectively. A plain configuration c is said to be *persistent* if each complete run from c is persistent from c (i.e., $\Pi^{\mathsf{Persistent}}(c) \subseteq \Pi^{\mathsf{Complete}}(c)$); otherwise it is called *fragile*. An instance of *the persistence* problem defined on a program $\mathcal{P}$ asks whether the initial configuration c_{init} is persistent or not. In Fig. 4, the run ρ_1 is persistent from the configuration d (of Fig. 9), while ρ_3 is fragile from d.

Safety Properties. We can show that if a program is persistent then it is *strongly persistent*. This means that, for any plain configuration c and run π, it is the case that $c_{init} \xrightarrow{\pi} c$ iff $c_{init} \xrightarrow{\pi'} c$ from some $\pi' \in \Pi^{\mathsf{SC}}$. In other words, c is reachable from c_{init} under TSO iff it is reachable under SC. It is well-known that checking safety properties can be expressed as reachability of sets of (plain) configurations. This implies that a persistent program satisfies the same safety properties under SC and TSO.

7 Fragility Pattern

We perform the first step in solving the persistence problem. We show that the persistence problem can be reduced to searching for runs of a special form More precisely, for a given plain configuration c, there is a fragile run from c iff there is another run from c that will follow a certain *fragility pattern*.

Fix a program $\mathcal{P} = \langle \mathbb{X}, \mathsf{ProcSet}, \mathbb{Q}, \mathbb{I}, init \rangle$. For a plain configuration c, a process $p \in \mathsf{ProcSet}$, a variable $x \in \mathbb{X}$, and a value $v \in V$, we define $\Pi^{\circledast}_{p,x,v}(c)$ to be the set of runs π such that $\pi = \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4$, and the following conditions are satisfied: (i) $\pi \in \Pi^{\mathsf{SC}}(c)$. (ii) $\pi_2 = e_1 \cdot u_1$, where $e_1 \in \Delta_p^{\mathsf{wr},y}$ and $u_1 \in \Delta_p^{\mathsf{ud},y}$ for some $y \neq x$. (iii) $\pi_3 \in \Pi\left(\Delta_p - \left(\Delta_p^{\mathsf{wr}} \cup \Delta_p^{\mathsf{ud}} \cup \Delta_p^{\mathsf{arw}} \cup \Delta_p^{\mathsf{fn}}\right)\right)$, i.e., π_3 consists only of events performed by p excluding write, update, atomic-read-write, and fence

events. (iv) $\pi_4 = e_2 \cdot u_2$, where $e_2 \in \Delta_r^{\mathsf{wr},x,v'}$ and $u_2 \in \Delta_r^{\mathsf{ud},x,v'}$ for some $r \neq p$ and $v' \neq v$. (v) There is an event $e \in \Delta_p^{\mathsf{rd},x,v}$, called the *complementary event* of π, such that $\pi_1 \cdot \pi_2 \cdot \pi_3 \cdot e \in \Pi\,(c)$, i.e., if we replace π_4 by e, then it results in a run from c. We call the process p the *pivot* of π, and call the rest of the processes the *fringe processes* of π. In §3, we motivate why $\rho_* \in \Pi^{\circledast}_{p_1,x,2}\,(d)$. We define $\Pi^{\circledast}\,(c) := \cup_{p \in \mathsf{ProcSet}} \cup_{x \in \mathbb{X}} \cup_{v \in V} \Pi^{\circledast}_p\,(c)$, and call $\Pi^{\circledast}\,(c)$ the set of runs of *type* $\circledast$ from c. We can show:

Theorem 1. *$\Pi^{\circledast}\,(c) = \emptyset$ iff $\Pi^{\mathsf{Fragile}}\,(c) = \emptyset$.*

In other words, to check whether c is fragile, we need only to check whether there is a run of type $\circledast$ from c. Although a run π of type $\circledast$ is SC, its existence together with the complementary event e show the fragility of c. More precisely, we define the *witness* run $\mathsf{Witness}\,(\pi) := \pi_1 \cdot e_1 \cdot \pi_3 \cdot e \cdot e_2 \cdot u_2 \cdot u_1$, and observe that $\mathsf{Witness}\,(\pi) \in \Pi^{\mathsf{Fragile}}\,(c)$. The run $\mathsf{Witness}\,(\pi)$ is TSO since e overtakes e_1. However, it is not persistent since there is no SC run with the same program and store order. In §3, we give the witness run corresponding to ρ_*.

Theorem 1 holds for any program respecting the syntax of §5. In particular, the program may contain any number of variables, and the variables may range over unbounded data domains (see §3, Model).

8 Pattern Detection

We perform the second step in solving the persistence problem. We translate the persistence problem for programs running under TSO to the SC reachability problem. We exploit Theorem 1 which shows that the persistence problem is reducible to the problem of checking whether the initial configuration has a type $\circledast$ run. Consider an instance of the persistence problem, defined by a program $\mathcal{P} = \langle \mathbb{X}, \mathsf{ProcSet}, \mathbb{Q}, \mathbb{I}, \mathit{init} \rangle$ and its initial configuration c_{init}. We will translate the problem of whether $\mathcal{P}$ has a run in $\Pi^{\circledast}\,(c_{\mathit{init}})$ to an instance of the SC reachability problem defined on a new program $\mathcal{Q} = \left\langle \mathbb{X}^{\mathcal{Q}}, \mathsf{ProcSet}^{\mathcal{Q}}, \mathbb{Q}^{\mathcal{Q}}, \mathbb{I}^{\mathcal{Q}}, \mathit{init}^{\mathcal{Q}} \right\rangle$. We define $\mathsf{ProcSet}^{\mathcal{Q}} := \mathsf{ProcSet}$, and $\mathit{init}^{\mathcal{Q}} := \mathit{init}$, i.e., $\mathcal{Q}$ uses the same set of processes as $\mathcal{P}$ and the processes start from identical initial labels. Each process p in $\mathcal{Q}$ will simulate the corresponding process in $\mathcal{P}$. Recall that a run $\pi \in \Pi^{\circledast}_{p,x,v}\,(c)$ is of the form $\pi = \pi_1 \cdot e_1 \cdot u_1 \cdot \pi_3 \cdot e_2 \cdot u_2$, and it is defined in terms of a pivot p, and a complementary event e. We know that π induces a witness (fragile run) $\mathsf{Witness}\,(\pi) = \pi_1 \cdot e_1 \cdot \pi_3 \cdot e \cdot e_2 \cdot u_2 \cdot u_1$. A run of $\mathcal{Q}$ is divided into three phases: $0, 1, 2$, where each phase will accomplish a particular task in the simulation of $\mathsf{Witness}\,(\pi)$. To carry out the simulation we add two new constants, $\mathfrak{c}_1$ and $\mathfrak{c}_2$ to the domain V, and add two new variables, ph with domain $\{-1, 0, 1, 2\}$, and new with domain V to the set of variables. In other words, we define $\mathbb{X}^{\mathcal{Q}} := \mathbb{X} \cup \{\mathsf{ph}, \mathsf{new}\}$. Furthermore, we define $\mathbb{Q}^{\mathcal{Q}} := \mathbb{Q} \cup \{q' \mid q \in \mathbb{Q}\} \cup \{\mathsf{final}_p \mid p \in \mathsf{ProcSet}\} \cup \mathbb{Q}^{\mathtt{tmp}}$, and define $\mathsf{Final} := \cup_{p \in \mathsf{ProcSet}} \{c \mid \mathtt{LabelOf}\,(c)\,(p) = \mathsf{final}_p\}$. Intuitively, for each label q in $\mathcal{P}$, we create a copy q' (that will be used to simulate the moves of the pivot).

Furthermore, we provide each process with an additional label that marks its "accepting state". Finally, $\mathbb{Q}^{\mathtt{tmp}}$ contains a set of "temporary labels", each of which is either of the form $q.i$ or $q'.i$, where $q \in \mathbb{Q}$, and $i \in \mathbb{N}$. To simplify the definition of $\mathcal{Q}$, we will extend the set of expressions (see §5) by allowing the use of shared variables in expressions. Since the program $\mathcal{Q}$ runs under the SC semantics, the evaluation $c(x)$ of a shared variable x in a configuration c is straightforward (it is given by $\texttt{MemoryOf}\,(c)\,(x)$).

Phase 0. Phase 0 corresponds to simulating π_1. During this phase, the processes will run the same code as in $\mathcal{P}$. A process needs to make sure that the program is currently running in phase 0. This is accomplished as follows. For each process $p \in \mathsf{ProcSet}$, and each instruction `q1: 𝔰; goto q2` in $\mathbb{I}_p$, we add the instructions of Fig. 10 to $\mathbb{I}_p^{\mathcal{Q}}$. First, the process changes the phase to -1 (thus blocking the moves of all other processes). In the next step, the process simulates the given instruction, after which it puts the phase back to 0 (thus unblocking the rest of the processes).

```
q1: arw(ph,0,-1); goto q1.1
q1.1: 𝔰; goto q1.2
q1.2: ph=0; goto q2
```

Fig. 10. Phase 0

At any point where a process p is about to execute a write event (on a variable y), it may decide to declare the event to be e_1. In this case, p will play the role of the pivot for the rest of the run, while deeming the other processes to become fringe processes. This signals the end of phase 0, and the phase is changed to 1. At the same time, we assign the value $\mathfrak{c}_1$ to y. The value of y will not be changed in the rest of the run, and y will be the only variable whose value is equal to $\mathfrak{c}_1$. Notice that, by definition of a type $\circledast$ run, y will not be used again in the rest of the run and therefore it is safe to change its value to $\mathfrak{c}_1$. To carry out the above steps, we add, for each write instruction $\mathfrak{ins} \in \mathbb{I}_p$, of the form `q1: y=𝔢; goto q2` the instructions shown in Fig. 11 to $\mathbb{I}_p^{\mathcal{Q}}$.

```
q1: arw(ph,0,-1); goto q1.3
q1.3: y=𝔠1; goto q1.4
q1.4: ph=1; goto q'2
```

Fig. 11. Change to phase 1

Phase 1. The purpose of phase 1 is to simulate the π_3. Since π_3 consists only of events performed by the pivot, only this process is active during phase 1. Furthermore, the pivot only performs read, skip, assignment, and assume events. For each instruction in $\mathbb{I}_p^{\mathsf{rd}} \cup \mathbb{I}_p^{\mathsf{skip}} \cup \mathbb{I}_p^{\mathsf{asgn}} \cup \mathbb{I}_p^{\mathsf{asm}}$, of the form `q1: 𝔰; goto q2`, we add the instructions of Fig. 12 to $\mathbb{I}_p^{\mathcal{Q}}$. Notice that, since only p is active during this phase, we need not lock the variable ph.

```
q'1: assume ph=1; goto q'1.1
q'1.1: 𝔰; goto q'2
```

Fig. 12. Phase 1

At any point where the pivot is about to execute a read event, it may decide to verify the existence of the complementary event e. Recall that $e \in \Delta_p^{\mathsf{rd},x,v}$, where $x \neq y$. For each read instruction $\mathfrak{ins} \in \mathbb{I}_p^{\mathsf{rd},x}$, of the form `q1: $𝔯=x; goto q2`, we add the instructions shown in Fig. 13 to $\mathbb{I}_p^{\mathcal{Q}}$. The instruction at q'1.2 checks that the read variable is different from y. Recall that

y is the only variable whose value is $\mathfrak{c}_1$. At q'1.3, we store the current value of x in new (this will be used in phase 2). At q'1.4, the value $\mathfrak{c}_2$ is stored in x. Notice that x is now the only variable whose value is $\mathfrak{c}_2$. By this assignment, the pivot has declared (i) that it was about to perform a read on x; and (ii) that the value it was about to read is stored in new. At this point, the pivot changes the phase to 2. This is the last instruction executed by the pivot. Observe that the process does not execute the instruction ins itself, but it marks its existence through the instructions of Fig. 13.

```
q'1: assume ph=1; goto q'1.2
q'1.2: assume !(x=c1); goto q'1.3
q'1.3: new=x; goto q'1.4
q'1.4: x=c2; goto q'1.5
q'1.5: ph=2; goto q'2
```

Fig. 13. Change to phase 2

Phase 2. In phase 2, only the fringe processes are active. The purpose of this phase is to verify the existence of the event e_2. This event is performed by a fringe process and it should write a value different from $\mathfrak{c}_2$ to x. A process that is about to execute a write event, may verify that this event corresponds to e_2. The process recognizes the variable x, since x is the only variable carrying the value $\mathfrak{c}_2$. For each write instruction of the form `q1: z=𝔠; goto q2` in $\mathbb{I}_p^{\mathsf{wr}}$, we add the instructions shown in Fig. 14 to $\mathbb{I}_p^{\mathcal{Q}}$. The test $z = \mathfrak{c}_2$ ensures that z and x are identical. The test $!(\mathsf{new} = \mathfrak{c})$ ensures that the current instruction assigns a value different from the value of x during the complementary event. In such a case, a witness has been found and the process moves to the accepting label final_p.

```
q1: assume ph=2; goto q1.5
q1.5: assume z=c2; goto q1.6
q1.6: assume !(new=𝔠); goto finalp
```

Fig. 14. Phase 2

Remarks. Notice that $\mathcal{Q}$ contains only two additional variables, namely new and ph, compared to $\mathcal{P}$; and that we increase the variable domain V by two elements, namely $\mathfrak{c}_1$ and $\mathfrak{c}_2$.

9 Fence Insertion

In this section we describe our fence insertion procedure that finds a minimal set of fences sufficient for making the program persistent. The algorithm builds a set of fences successively using fragile runs generated by the pattern detection algorithm. First, we define the fence insertion operation, and then show how to use a type $\circledast$ run generated by the pattern detection algorithm of §8 to derive a set of fences such that the insertion of at least one element of the set is necessary in order to eliminate the run from the behavior of the program. Based on that, we introduce the fence insertion algorithm.

Fence Insertion. We define the operation of inserting a fence in a program. Intuitively, we identify the instruction after which we insert the fence. For a program $\mathcal{P} = \langle \mathbb{X}, \mathsf{ProcSet}, \mathbb{Q}, \mathbb{I}, \mathit{init} \rangle$ and an instruction $\mathfrak{f} \in \mathbb{I}$, we use $\mathcal{P} \oplus \mathfrak{f}$ to

denote the program we get by inserting a fence instruction just after $\mathfrak{f}$ in $\mathcal{P}$. Formally, let $\mathfrak{f} \in \mathbb{I}$ be of the form $q_1 : \mathfrak{s}; \texttt{goto}\ q_2$. Then, $\mathcal{P} \oplus \mathfrak{f}$ is the program we get by replacing $\mathfrak{f}$ by the following two instructions (where $q' \notin \mathbb{Q}$ is a unique new label): $q_1 : \mathfrak{s}; \texttt{goto}\ q'$, and $q' : \mathsf{fn}; \texttt{goto}\ q_2$ (recall from §5 that fn is the fence statement). For a set $F = \{\mathfrak{f}_1, \ldots \mathfrak{f}_n\} \subseteq \mathbb{I}$, we define $\mathcal{P} \oplus F := \mathcal{P} \oplus \mathfrak{f}_1 \cdots \oplus \mathfrak{f}_n$. We say F is *minimal* wrt. $\mathcal{P}$ if (i) $\mathcal{P} \oplus F$ is persistent, and (ii) $\mathcal{P} \oplus (F \setminus \{\mathfrak{f}\})$ is fragile for all $\mathfrak{f} \in F$. That is, removing any fence from F makes $\mathcal{P}$ fragile.

Fence Inference. Let $p \in \mathsf{ProcSet}$, and let π be a type $\circledast$ run generated by applying an arbitrary SC reachability analysis algorithm on the program $\mathcal{Q}$ defined in §8. Recall from §7, that from π, we can derive a new run $\mathsf{Witness}(\pi)$ of the form $\pi_1 \cdot e_1 \cdot \pi_3 \cdot e \cdot e_2 \cdot u_2 \cdot u_1$, then $\mathsf{Witness}(\pi) \in \Pi^{\mathsf{Fragile}}(c)$, where e overtakes e_1. Let π_3 be of the form $e'_1 e'_2 \cdots e'_n$, and define $\mathsf{NewFences}(\pi) := \{e_1, e'_1, e'_2, \ldots, e'_n\}$. Intuitively, inserting a fence after one of the instructions in $\mathsf{NewFences}(\pi)$ is both necessary and sufficient to prevent e from overtaking e_1, and hence eliminating the run π from the behavior of the program.

Algorithm. We present our fence insertion algorithm (Algorithm 1). It takes a concurrent program $\mathcal{P}$ and returns a minimal set of fences that is sufficient to make $\mathcal{P}$ persistent. The algorithm uses a set of sets of fences, namely $\mathcal{W}$ for sets of fences that have been *partially* constructed (but not yet large enough to make the program persistent). During each iteration, a set F is picked and removed from $\mathcal{W}$. We use the construction of §8, together with an SC reachability analysis algorithm, to check whether the set F is sufficient to make the program persistent. If *yes*, we return F as a possible set of minimal fences. If *no*, we compute the set of fences N such that inserting a member of N is sufficient and necessary to eliminate the generated type $\circledast$ run π. For each $\mathfrak{f} \in \mathsf{N}$ we add $F' = F \cup \{\mathfrak{f}\}$ back to $\mathcal{W}$ unless there is already a subset of F' in the set $\mathcal{W}$.

Algorithm 1. Fence Insertion.

```
   input  : A concurrent program P
   output: A minimal set of fences
1  W ← {∅};
2  while true do
3      Pick and remove a set F from W;
4      if ∃π : π ∈ Π_p^⊛ (c_init) in P ⊕ F then
5          N ← NewFences(π);
6          foreach f ∈ N do
7              F' ← F ∪ {f};
8              if ∃F'' ∈ W : F'' ⊆ F' then discard F' else W ← W ∪ {F'}
9      else
10         return F;
```

Theorem 2. *If each call to the pattern detection algorithm (line 4) returns, then Algorithm 1 terminates and returns a minimal set of fences wrt. $\mathcal{P}$.*

In particular, Theorem 2 implies that Algorithm 1 terminates when $\mathcal{P}$ is a finite-state program.

In the program of Fig. 2, our algorithm would insert a single fence, replacing the instruction q2. z=1; goto q3 in p_1 by the instructions q2. z=1; goto q5 and q5. fn; goto q3.

10 Observation Abstraction

In this section, we present a general abstraction framework, called *observation abstraction*, that is compatible with the notion of persistence (*compatibility* means that persistence of the abstract program implies persistence of the concrete program). The abstraction considers a process p and captures the sequences of events that can be observed from p. We instantiate observation abstraction by defining an abstraction function, namely *Flattening*, whose efficiency is demonstrated in the experimental results (see §11). Let us fix a program $\mathcal{P} = \langle \mathbb{X}, \mathsf{ProcSet}, \mathbb{Q}, \mathbb{I}, \mathit{init} \rangle$ and a process $p \in \mathsf{ProcSet}$. Let $R := \mathsf{ProcSet} \setminus \{p\}$.

Notation. We define $\mathsf{ReadFrom}_p := \left\{ x \in \mathbb{X} \mid \left(\Delta_p^{\mathsf{rd},x} \cup \Delta_p^{\mathsf{arw},x} \right) \neq \emptyset \right\}$, i.e., it is the set of shared variables on which p may perform read or atomic-read-write events. We define a new type of events, namely $\mathsf{MemEvent}(x, v)$, where $x \in \mathbb{X}$ and $v \in V$, that we use to abstract update and atomic-read-write events. The event records changes in the state of the memory (changing the value of x to v), while hiding the identity of the process performing the event. For a variable $x \in \mathbb{X}$, a value $v \in V$, an event $e \in \Delta^{\mathsf{arw},x,v} \cup \Delta^{\mathsf{ud},x,v}$, and a process $p \in \mathsf{ProcSet}$, we will write $[e]_p$ to describe "how much of e can be observed by p". Formally: (i) $[e]_p := e$ if $e \in \Delta_p$, i.e., we keep the event if it is performed by p (p can observe its own events). (ii) $[e]_p := \mathsf{MemEvent}(x, v)$ if $e \in \Delta_r^{\mathsf{arw},x,v} \cup \Delta_r^{\mathsf{ud},x,v}$, for some $r \neq p$ and $x \in \mathsf{ReadFrom}_p$. If the event is performed by another process on a variable in $\mathsf{ReadFrom}_p$ then p can observe the change in memory although it cannot see the process making the change. (iii) $[e]_p := \epsilon$ if $e \in \Delta_r^{\mathsf{arw},x} \cup \Delta_r^{\mathsf{ud},x}$, for some $r \neq p$ and $x \notin \mathsf{ReadFrom}_p$. The event is not observed by p in case it is performed by another process on a variable not in $\mathsf{ReadFrom}_p$. For a run $\pi = e_1 e_2 \cdots e_n \in \left(\Delta^{\mathsf{arw},x} \cup \Delta^{\mathsf{ud},x} \right)^*$, we define $[\pi]_p := [e_1]_p [e_2]_p \cdots [e_n]_p$.

Framework. In addition to (the concrete program) $\mathcal{P}$ with initial configuration c_{init}, we consider an abstract program $\mathcal{A}_p = \left\langle \mathbb{X}^{\mathcal{A}_p}, \mathsf{ProcSet}^{\mathcal{A}_p}, \mathbb{Q}^{\mathcal{A}_p}, \mathbb{I}^{\mathcal{A}_p}, \mathit{init}^{\mathcal{A}_p} \right\rangle$, with initial configuration $c_{\mathit{init}}^{\mathcal{A}_p}$. We assume that the following conditions hold: (i) $\mathbb{X}^{\mathcal{A}_p} = \mathbb{X}$, i.e., $\mathcal{P}$ and $\mathcal{A}_p$ operate on the same set $\mathbb{X}$ of shared variables. (ii) The process p is in $\mathsf{ProcSet}^{\mathcal{A}_p}$. (iii) $\mathbb{I}_p = \mathbb{I}_p^{\mathcal{A}_p}$, i.e., the process p executes the same code in $\mathcal{P}$ and $\mathcal{A}_p$. We say that $\mathcal{A}_p$ is an observation abstraction of $\mathcal{P}$ wrt. p, denoted $\mathcal{P} \sqsubseteq_p \mathcal{A}_p$, if for every run $\pi \in \Pi_p(c_{\mathit{init}})$ in $\mathcal{P}$, there is a run $\pi' \in \Pi_p\left(c_{\mathit{init}}^{\mathcal{A}_p}\right)$ in $\mathcal{A}_p$ such that the following two conditions are satisfied: (i) $\mathsf{ProgOrder}(\pi)(p) = \mathsf{ProgOrder}(\pi')(p)$. (ii) $[\mathsf{StoreOrder}(\pi)]_p = [\mathsf{StoreOrder}(\pi')]_p$. In other words, the programs $\mathcal{P}$ and $\mathcal{A}_p$ agree on the "parts of runs" that are observable by p: on the one hand, p observes all events it performs itself; on the other hand, it observes modifications of the memory performed by other processes provided that they concern variables from which it can read (in the latter case, the actual identity of the process performing the event is not relevant).

The next theorem shows that persistence can be established by analyzing the abstract programs.

Theorem 3. *For every $p \in \mathsf{ProcSet}$, let $\mathcal{A}_p$ be an abstract program such that $\mathcal{P} \sqsubseteq_p \mathcal{A}_p$. If $\mathcal{A}_p$ is persistent for all $p \in \mathsf{ProcSet}$ then $\mathcal{P}$ is persistent.*

Flattening. We define a family of functions that build observation abstractions $\mathcal{A}_p$ of $\mathcal{P}$ wrt. p for $k \in \mathbb{N}$. Flattening keeps all processes in $\mathcal{P}$ applying abstraction individually on each process. Although the number of processes remains the same, the abstraction simplifies the behavior of each process, again limiting the state explosion problem. The precision of the abstraction increases with increasing the value of k. More precisely, we keep the behavior of a process r during its first k steps after which we replace each statement, except write and atomic-read-write statements on variables in $\mathsf{ReadFrom}_p$, by the empty statement. Flattening abstraction is precise in the sense that it does not add extra fences (compared to the set of fences that would be added to the concrete program). In our experiments, no additional fences are added when applying this abstraction.

```
process p1
regs  init s
begin
 q1.  t=1; goto q2
 q2.  skip; goto q2
 q2.  skip; goto q3
 q3.  skip; goto q4
end
```

Fig. 15. Flattening abstraction of the program of Fig. 2 wrt. p_2

11 Experimental Results

Tool. We have implemented our techniques from §8-§10 in an open-source tool called Persist. The syntax of the input language of Persist is defined by the grammar of Fig. 8 in §5. Persist uses Spin [16] as backend model-checker to solve the SC reachability queries for programs under SC. Since Spin is a finite-state model checker, all the programs in our experiments are finite-state. We compare our method with state-of-the-art tools[1]: Trencher [9] (a sound and complete tool for robustness analysis of finite-state programs under TSO, that uses Spin as backend tool), Memorax [1] (a sound and complete tool for the correctness analysis of finite-state programs under TSO wrt. safety properties), Remmex [20] (a tool based on state-space exploration with acceleration, for correctness analysis of programs under TSO wrt. to safety properties), and Musketeer [3] (a static analysis tool for correctness analysis of programs under weak memory models wrt. robustness). We perform the comparisons based on two aspects, namely the *number of fences* (and their placement) and the *running time*. The experiments are performed on an Intel x86-32 Core2 2.4 Ghz machine and 4GB of RAM. To insert the fences, Memorax and Remmex require a safety property as an additional input which is not always given; while Persist, Trencher, and Musketeer are fully automatic. Notice that both persistence and robustness guarantee SeqCon.

In the following, we present two sets of results. The first set concerns the comparison of Persist (without the abstraction) with the other tools (see Table 1). The second set shows the scalability of Persist (with/without abstraction) compared to Musketeer and Trencher when increasing the number of processes (see

[1] Except Dfence [21] which requires a special manual specification encoding for each example.

Fig. 16). In all experiments, we set up the time out to 2400 seconds, and $k = 2$ for abstraction. Our examples are from [21,9,1,20,3].

Performance of Persist *without abstraction.* The results are given in Table 1. Below, we summarise the main observations: (i) Persist manages to return for all the benchmarks. Trencher and Musketeer fail to answer for 8 out of 35 examples within 2400 seconds. Memorax (Remmex) manages to return for only 4 (9) out of 30 examples within 2400 seconds. For the comparison among tools, we compute the average of the ratio of the running times over examples where the tools terminate. On average Persist is 51 times faster than Trencher and 2.48 times faster than Musketeer. (ii) Persist returns 54 fences while Musketeer returns 119 fences for all the examples where Musketeer terminates within 2400 seconds (and thus, Persist inserts 54% less fences in total). Furthermore, Persist returns 44 fences while Trencher returns 62 fences for all the examples where Trencher terminates within 2400 seconds (and thus, Persist inserts 30% less fences in total).

Table 1. Experimental results. Per, Tre, Mus, Mem, and Rem stand for Persist, Trencher, Musketeer, Memorax, and Remmex, respectively. The columns #P, #F, and #T give the number of processes, number of fences, and running time in seconds, respectively. If a tool runs out of time (resp. memory), we put "TO" (resp. "OM") in the #T column, and • in #F column. We use "–" when a tool is not tested due to a missing specification.

Program	#P	Per		Tre		Mus		Mem		Rem	
		#F	#T	**#F**	#T	**#F**	#T	**#F**	#T	**#F**	#T
SimDekker	4	**4**	4	**4**	163	**8**	1	•	OM	•	OM
Dekker	4	**8**	14	•	TO	**16**	783	•	OM	•	OM
Peterson	4	**4**	223	•	TO	•	TO	•	OM	•	OM
LamBak	3	**6**	104	•	TO	**18**	110	•	OM	**6**	372
LamBak	4	**8**	286	•	TO	•	TO	•	OM	•	OM
Dijkstra	4	**8**	30	•	TO	•	TO	•	OM	•	OM
Dc-Lock	6	**0**	7	**0**	139	**0**	1	•	TO	**0**	5
SpinLock	2	**0**	1	**0**	1	**0**	1	**0**	1	**0**	1
TSpinLock	2	**0**	1	**0**	1	**0**	1	•	OM	**0**	4
InlinePgsql	8	**0**	8	**0**	426	•	TO	•	OM	**0**	42
Burns	5	**9**	119	•	TO	•	TO	•	OM	•	TO
Szymaski	2	**8**	3	**8**	642	**11**	1	**3**	1	**3**	4
Szymaski	4	**16**	88	•	TO	•	TO	•	OM	•	TO
LamFast2	2	**8**	6	**4**	48	**15**	1	**4**	136	**4**	6
LamFast2	3	**12**	10	•	TO	•	TO	•	OM	**6**	108
CLH Lock	4	**4**	176	**4**	674	•	TO	•	OM	•	OM
Parker	6	**1**	76	**1**	425	**3**	1	•	OM	•	OM
Pgsql	6	**7**	26	**7**	260	**7**	1	•	OM	•	TO
AltenatingBit	2	**4**	2	**4**	5	**5**	1	**0**	4	**0**	3
IncSequence	6	**0**	21	**0**	194	**0**	1	•	OM	•	TO
TaskSchedule	10	**0**	5	**0**	418	**0**	1	•	TO	•	OM
NBW_W_WR	2	**0**	18	**1**	595	**6**	1	•	OM	•	TO
NBW_W_5R	6	**0**	3	**0**	514	**0**	1	•	OM	•	TO
SeqLock	4	**0**	63	**0**	364	**0**	1	•	OM	•	TO
write+r	5	**0**	2	**4**	7	**4**	1	•	OM	•	TO
r+detour	5	**0**	1	**3**	3	**3**	1	•	OM	•	TO
r+detours	5	**0**	1	**3**	1	**3**	1	•	OM	•	TO
write+r+co	6	**0**	7	**4**	38	**4**	1	•	OM	•	TO
sb+detours	6	**2**	3	**5**	3	**5**	1	•	OM	•	TO
sb+detour+co	6	**0**	1	**4**	3	**4**	1	•	OM	•	TO
Cilk WSQ	2	**2**	3	**2**	107	**3**	1	–	–	–	–
CL WSQ	2	**1**	2	**1**	796	**1**	1	–	–	–	–
FIFO iWSQ	2	**1**	2	**1**	354	**1**	1	–	–	–	–
LIFO iWSQ	2	**1**	7	**1**	558	**1**	1	–	–	–	–
Anchor iWSQ	2	**1**	2	**1**	20	**1**	1	–	–	–	–

Scalability wrt. the number of processes. We compare the scalability of Trencher, Musketeer, Persist, and our abstraction Flattening) while increasing the number of processes in several examples. In all these examples, no additional fences are added due to flattening abstraction (compared to the case of Persist without abstraction). The results are given in Fig. 16. We observe that: (i) Persist without

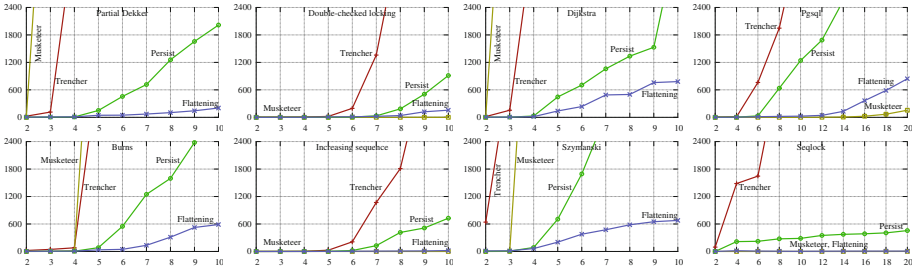

Fig. 16. Persist with/without abstraction compared with Trencher and Musketeer. The x axis is number of processes, the y axis is running time in seconds.

abstraction always scales better than Trencher for all the examples. Furthermore, Persist without abstraction scales better than Musketeer in 4 out of 8 examples (Partial Dekker, Burns, Dijkstra, Szymanski). (ii) Persist with the abstraction always performs better than Persist without the abstractions while inserting no extra fences. (iii) Persist with the abstraction outperforms Musketeer in 4 out of 8 examples (Partial Dekker, Burns, Dijkstra, Szymanski) while having comparable running time for the rest.

In all the reported results (Table 1 and Fig. 16), except *Lamport Fast 2 with two processes (LamFast2)*, the set of fences returned by Persist is a subset of the ones returned by Trencher and Musketeer. Hence, Persist presents a good trade-off between efficiency and optimality.

12 Conclusion, Discussion, and Future Work

We have presented a framework for automatic fence insertion under TSO that provides an excellent trade-off between efficiency and optimality. We have implemented our framework in a tool and evaluated it on a wide range of benchmarks. The correctness criteria of Fig. 1, namely Data Race Freedom (Drf), Triangular Race Freedom (Trf), Robustness (Rob), Persistence (Per), Sequential Consistence (SeqCon), and state Reachability (Reach) can be seen as "stability conditions", in the sense that they measure how stable the behaviors of the program is under TSO compared to SC. Only a small number of stability conditions are relevant, since each condition corresponds to relaxing one of three parameters: program order, source, and (variable) store order. A stability condition should imply that the program under SC and TSO have the same reachable (control) states, and that they satisfy the same safety properties (see §1, Persistence; and §6, Safety Properties). We believe that our work is an important step in this investigation. There are several open and hard questions to consider in future work (see Fig. 17). This includes studying two remaining important stability conditions, namely PoVso where a program is considered to be correct if the traces

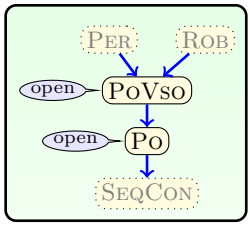

Fig. 17. Relevant correctness criteria

of the program under TSO and SC agree on (i) program order, and (ii) variable store (coherence) order. PoVso gives an entirely different stability condition compared to persistence (in fact, PoVso is a weakening of both robustness and persistence). Checking PoVso (e.g., through finding appropriate patterns) is an important (and difficult) open problem. Notice that, if persistence and PoVso were equivalent, then robustness would be stronger than persistence which is not the case (they are incomparable). Another open problem is checking the condition Po, a weakening of PoVso, where the program is considered if its TSO and SC traces need only to agree on program order. Finally, it is important to develop frameworks that allow checking the different stability conditions for other weak memory models, as done in [4].

Acknowledgment. The authors would like to thank Jade Alglave, Ahmed Bouajjani, Roland Meyer, Madan Musuvathi and Viktor Vafeiadis for helpful discussions. We thank Madan Musuvathi again for pointing us to some benchmarks. We also thank Carl Leonardsson, Jade Alglave, Egor Derevenetc, Daniel Kroening, Alexander Linden, Roland Meyer, and Martin Vechev for giving us access to their tools and their benchmarks.

References

1. 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)
2. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer 29(12), 66–76 (1996)
3. Alglave, J., Kroening, D., Nimal, V., Poetzl, D.: Don't sit on the fence. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 508–524. Springer, Heidelberg (2014)
4. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013)
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. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL 2010, pp. 7–18. ACM (2010)
7. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: What's decidable about weak memory models? In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 26–46. Springer, Heidelberg (2012)
8. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)
9. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013)

10. 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)
11. 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)
12. 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)
13. Duan, Y., Muzahid, A., Torrellas, J.: Weefence: toward making fences free in TSO. In: Mendelson, A. (ed.) ISCA 2013, pp. 213–224. ACM (2013)
14. Fraser, K.: Practical lock-freedom. Tech. Rep. UCAM-CL-TR-579, University of Cambridge, Computer Laboratory (2004)
15. Gharachorloo, K., Gupta, A., Hennessy, J.: Performance evaluation of memory consistency models for shared-memory multiprocessors. SIGPLAN Not. 26(4), 245–257 (1991)
16. Holzmann, G.: SPIN Model Checker, the: Primer and Reference Manual, 1st edn. Addison-Wesley Professional (2003)
17. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. In: Bloem, R., Sharygina, N. (eds.) FMCAD 2010, pp. 111–119. IEEE (2010)
18. Kuperstein, M., Vechev, M.T., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: Hall, M.W., Padua, D.A. (eds.) PLDI 2011, pp. 187–198. ACM (2011)
19. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: Arora, S., Leavens, G.T. (eds.) OOPSLA 2009, pp. 227–242. ACM (2009)
20. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in relaxed memory systems. In: Groce, A., Musuvathi, M. (eds.) SPIN Workshops 2011. LNCS, vol. 6823, pp. 144–160. Springer, Heidelberg (2011)
21. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: Vitek, J., Lin, H., Tip, F. (eds.) PLDI 2012, pp. 429–440. ACM (2012)
22. Marino, D.L.: Simplified Semantics and Debugging of Concurrent Programs via Targeted Race Detection. Ph.D. thesis, University of California at Los Angeles, Los Angeles, CA, USA (2011)
23. Michael, M.M., Vechev, M.T., Saraswat, V.A.: Idempotent work stealing. In: Reed, D.A., Sarkar, V. (eds.) PPOPP 2009, pp. 45–54. ACM (2009)
24. 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)
25. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009)
26. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7), 89–97 (2010)
27. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. 10(2), 282–312 (1988)