Ensuring The Correctness of Concurrent Programs under TSO Memory Models

Tuan-Phong NGO
Abstract

Ensuring The Correctness of Concurrent Programs under TSO Memory Models

Tuan-Phong NGO

For efficiency reasons, most modern processor architectures allow the reordering of CPU instructions, resulting in weak memory models. These models add extra program executions that are not intended by the programmer, often causing subtle run-time errors. To help solve this problem, such architectures also provide memory fences that allow to eliminate undesired behaviors. However, manual fence insertion, is a tedious and time-consuming activity, that also needs to be repeated each time the program is updated. Therefore, the development of efficient tools for automatic fence insertion is a crucial challenge in concurrent program design.

In this thesis, we present, for the first time, a tool for automatic fence placement that is able to break the scalability barrier both concerning the added complexity due to the presence of event reorderings, and also concerning the number of threads that participate in the execution of the program. To this end, we propose a novel notion of correctness for concurrent programs, called persistence, that compares the behavior of the program under the weak memory semantics with that under the classical interleaving semantics.

To make our ideas concrete, we consider the Total Store Ordering (TSO) memory model, and show how our method (i) allows modular reasoning that limits state space explosion due to the presence of parallel processes (threads), and (ii) abstracts away complex behaviors caused by weak memory models by translating the problem, in linear time, into a verification problem that is defined under the interleaving semantics. We have implemented a prototype and run it successfully on all standard benchmarks, together with several challenging examples that are beyond the capability of existing methods.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td></td>
<td>iv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Motivation and Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Problem Statement</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Thesis’s Aim and Objectives</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Proposed Solution and Results</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>Related Work</td>
<td>7</td>
</tr>
<tr>
<td>1.6</td>
<td>Thesis Structure</td>
<td>8</td>
</tr>
<tr>
<td>2 Concurrent Programs</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>Preliminaries</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Concurrent Programs</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Process’s Automata</td>
<td>11</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Concurrent Programs</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Total Store Order Semantics</td>
<td>13</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Sequential Paths</td>
<td>13</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Total Store Order Rewriting</td>
<td>14</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Reordered Paths</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Program Runs</td>
<td>15</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Shuffling Operation</td>
<td>15</td>
</tr>
</tbody>
</table>
2.4.2 Sequential Runs ........................................... 16
2.4.3 Reorderd Runs ........................................... 17

3 Persistence ....................................................... 19
3.1 Traces ........................................................... 19
  3.1.1 Program Order .......................................... 19
  3.1.2 Total Store Order ....................................... 20
  3.1.3 Traces ...................................................... 20
  3.1.4 Persistence ............................................... 20
  3.1.5 Avoiding Bad Behaviors with Persistence ............... 20
  3.1.6 Minimal Fragility ........................................ 21
3.2 Solving Persistence ........................................... 21
  3.2.1 Type $\oplus$ Runs ....................................... 21
  3.2.2 Type $\ominus$ Runs ...................................... 22
  3.2.3 Type $\ominus$ Runs ...................................... 24
  3.2.4 Type $\oplus$ Runs ....................................... 26
  3.2.5 Type $\ominus$ Runs ...................................... 27
  3.2.6 Combination .............................................. 29
3.3 Code-to-Code Translation ................................... 30
3.4 Fence Insertion .............................................. 35
  3.4.1 Dangerous Attacks ...................................... 35
  3.4.2 Dangerous Paths ........................................ 36
  3.4.3 Algorithm ................................................ 36
  3.4.4 Integer Linear Programming ............................. 37
3.5 Compositional Verification .................................. 38
  3.5.1 Definitions and Notations ............................... 38
Acknowledgements

I would like to thank my supervisor Dr. Mohamed Faouzi Atig, my reviewer Pr. Parosh Aziz Abdulla, and my examiner Dr. Ivan Christoff at IT department, Uppsala University for their guidance and feedback when doing this thesis. The encouragement from my family and my girl friend, Quynh Anh, is also very important for me. When reading this thesis, if you find any mistake, sending to me at tuan-phong.ngo@it.uu.se is appreciated.
In this section, we describe the aim and objectives of our work, why it is hard and important. First we start with motivation and background, then we state our verification problem. Next we summarize our solution and results. Two final sections are dedicated for related work and thesis’s structure.

1.1 Motivation and Background

Concurrent programming and multi-core architectures [15] are very popular today. Concurrent programming that allows many processes (threads) to be alive and access to a same memory location at the same time can be executed both in single-core and multi-core architectures. In a single-core machine, operation system supports switching between processes; then at a cycle of CPU, there exactly only one process occupying and controlling the CPU core. Meanwhile in multi-core architectures, at a cycle each process can occupy one core, and there are many processes run in parallel.

Sequential consistent (SC) memory model [21] is the most simple and strictest model of multi-core architectures. Informally, SC semantics require that all operations are atomic. For example a write operation to a shared variable $x$ after issued
by a process will immediately updates its value to $x$ in the memory. Therefore all operations of a process appear in the order specified by its program's code, and execution a concurrent program appears as the same as interleaving instructions of processes in a sequential machine. Although SC model is very easy to understand, it prevents compilers and hardwares from many optimizations. Intuitively, suppose that in Figure 1.1 the write operation needs 1 cycle to be issued and 3 cycles to be completely updated to $x$ in memory; and each read operation of $y, z, t$ needs only 1 cycle to access its variable. By SC semantics, a process $p$ must spend 7 cycles to execute 4 instructions in sequence. However, from view of $p$, it can reduce the execution time to 4 cycles by issuing the write operation, not waiting its completely updated but executing 3 remaining read operations.

| 1: w(x,1); | 2: r(y,0); |
| 3: r(z,0); | 4: r(t,0); |

Figure 1.1: An optimizable code under weak memory models.

From the above example, we see that modern processor architectures can use weak memory models (WMM) [2] in which the scheduling of CPU instructions is governed by the availability of their operands rather than by the order in which they are issued [34]. In other words, WMM relaxes the order of instructions specified by program code to get a better instruction scheduling and hence increase the system performance. Each specific WMM may relax some of the following three instruction orders in a program code: write-read order, write-write order, and read-read/write order. However because of the relaxation, WMM can add more behaviors to a program than its behaviors under the SC semantics. This makes debugging, testing, and verification of concurrent programs under WMM become more difficult, and even undecidable [4].

The most common relaxation of WMM corresponds to Total-Store-Ordering mem-
ory model (TSO) that is adopted by Sun’s SPARC multiprocessors [36]. TSO is the kernel of many common weak memory models and is the latest formalization of the x86-tso memory model [29, 32]. TSO supports write-read relaxation by using a buffer for each process, allowing a read operation can be reordered before a write operation (reordering) or a read operation can read its own write early (read own write). For example, in Figure 1.2, process $p_2$ stores 4 write operations to $y$ in its buffer, then read $x$ directly from the memory. Because the write operations are not updated to memory, the read operation that is issued after write ones appears before the write operations. However, process $p_1$ has a write operation to $x$ in buffer, then if $p_1$ wants to read $x$, it must get value of $x$ from its buffer. Another example of TSO is the optimization we have seen in Figure 1.1.

Although WMM allows many optimizations of concurrent programs, moving a correct program under SC to WMM directly can generate non-intended behaviors. In fact, several algorithms such as mutex exclusion algorithms and producer-consumer protocols are not correct under WMM [2]. Figure 1.3 is a simplified version of Dekker’s mutex algorithm [12] that has two process $p_1$ and $p_2$, and two shared variables $x$ and $y$. When $p_1$ or $p_2$ wants to enter its critical section, it raises its flag $x$ or $y$ to 1 respectively, then checks whether the other process has already been in critical
section or not. Finally if $p_1$ knows that $p_2$ is not in critical section, $p_1$ can enter its critical section. We have the same rules for $p_2$. By setting and checking flags, Dekker algorithm guarantees that under SC semantics there is at most one process in critical section at a time. However, consider this algorithm under TSO model with a scenario as in Figure 1.4. At step $a$), both processes are in initial states. Next at step $b$), they raise their flags and store write operations to buffers. Then at step $c$) both $p_1$ and $p_2$ check and think that the other is not in critical section. Finally at step $d$) buffers are committed to memory and both processes are in critical sections. As consequence of this execution, simplified Dekker algorithm fails under TSO, allowing both processes in critical sections at the same time.

One way to eliminate the undesired behaviors is to insert memory fence instructions in program code. A fence instruction, executed by a process, implies that no reordering is allowed between instructions issued before and after the fence instruction. However, fence instruction is harmful to the system performance because it prevents compilers and hardwares to do optimizations. For example, in TSO model we can put a fence before each read operation or after each write operation to avoid non-intended behaviors. But by this way, this would result in a performance degra-
dation as it would mean that we would get closer to the SC model. This is an example of putting too much fences, and it is called over fencing. In other hand, if we insert too few fences (under fencing), it would result in unsound program behaviors. Therefore finding a minimal fence set has an important role for the efficiency of concurrent programs.

1.2 Problem Statement

From now we use non-intended behaviors or bad behaviors to indicate all behaviors of a concurrent program that are feasible under a specific WMM but infeasible under SC model. Verification problem of concurrent programs under WMM contains two aspects:

- *Input a concurrent program and a WMM, how to detect bad behaviors of this program?*

- *Input a concurrent program that contains bad behaviors under a WMM, how to find a minimal fence set to correct it?*

1.3 Thesis’s Aim and Objectives

This thesis considers the verification problem of concurrent programs under TSO model, including detecting bad behaviors and correcting them by a minimal fence set. The families of concurrent programs studied here include many practical algorithms such as mutex exclusion algorithms, producer-consumer protocols, and concurrent algorithms with dynamic structures (stack and queue). This thesis needs to propose a methodology and then develop a tool to solve the verification problem.
1.4 Proposed Solution and Results

Verifying bad behaviors (or safety errors) can be answered by solving a reachability problem where we determine from the initial state whether under WMM we can get some target states that cannot be reached under SC model. In fact the reachability problem under TSO is decidable but highly complex (non-primitive recursive complexity) [4]. Executing a program under SC or WMM generates a set of traces. Each trace has 3 orders: (i) program order: the order of commands issued by one process, (ii) total store order: the order of memory updates, and (iii) source relation that indicates the write command that a read receives its value from.

For the first time, we define a definition of persistence of programs. A program is persistent if and only if the traces of the program under SC and TSO have the same orders of program order and total store order. Keeping programs persistent guarantees that we can avoid all bad behaviors of the programs under TSO.

Based on the definition of persistence, an algorithm for automatic fence insertion is proposed that relies on code-to-code translation to a set of target programs each of which is linear in the size of the source program. The target program can be analyzed under the SC semantics. This allows for a very efficient fence insertion algorithm for the source program. Moreover to make the method scalable to programs with large numbers of processes, we introduce a sound thread abstraction technique that allows compositional reasoning and thus significantly limits the state explosion problem.

Finally we implement a tool, Persist, that we use to evaluate our framework on a wide range of benchmarks and real world examples, including several ones that are beyond the capabilities of existing methods.
1.5 Related Work

Figure 1.5 shows different stability criteria that have been used in the literature, including the one proposed in this paper, ordered according to their strength. The strongest condition is that of data race freedom (Drf) where the program is declared incorrect if it contains a trace with a data race. While checking this condition can be performed efficiently, it will cause severe over-fencing, since it amounts to the naive approach where we insert a fence instruction after every write instruction, or before every read instruction. This would mean that we get back to the SC model. In fact, many data races are in reality not harmful. For instance, lock-free data structures, transactional memories, and synchronization libraries sometimes explicitly employ data races although they rely on SC semantics for correctness. In view of this, more precise techniques, based on weaker conditions, have been developed to uncover real violations. In [28], triangular race freedom (Trf) is introduced where a program is considered to be correct if the traces of the program under the TSO and SC semantics agree on program order, total store order, and source relation. The main limitation of the Trf approach is that it does not come with a method for checking program correctness, and hence there is no known algorithm for fence insertion. Our approach is a weakening of the Trf condition in the sense we have removed the source relation, and therefore the latter will cause over-fencing compared...
to our method. Another weakening of the TRF condition is Robustness [33], where the total store order condition is replaced by a weaker condition, namely variable store order. The latter considers the order of memory updates performed on each variable individually. In [6], a tool based on code-to-code translation is provided for checking the robustness condition. Our translation produces a smaller program compared to that in [6], and hence, for a given set of processes, our method is more efficient. Also, crucially, no technique is known for compositional reasoning when checking the robustness condition, so the method does not scale for large numbers of processes. The method of [23] considers an even weaker condition. More precisely, it considers sequential consistence, i.e., it checks whether there are any states of the program that are reachable under the weak memory semantics, but not under the SC semantics. The weakest condition is considered in [1], where the method checks whether a given state of the program is reachable under the weak memory semantics. In contrast to our approach (and that of [6]) the frameworks of [1] do not provide code-to-code translation to the SC case. In fact, it is known that the correctness problem in both cases has a non-primitive recursive complexity. Therefore, [23] considers monitoring rather than verification, and [1] does not scale to the class of programs we consider in this paper.

1.6 Thesis Structure

The organization of this thesis is as follows. Chapter 2 defines definitions of process’s automata, concurrent programs, the Sequential Consistent semantics, and the Total-Store-Order semantics. Chapter 3 defines our definition of persistence based on program order and total-store order. Then it describes a specific pattern for non-persistent programs. We show that a concurrent program is non-persistent iff there exists a run with our specific pattern when running the concurrent program. This chapter also describes a code-to-code translation by which we transform our persis-
tence problem under TSO semantics to a verification problem under SC semantics. An algorithm to find a minimal fence set and compositional verification techniques are provided at the end of the chapter. Experiment result contains many tests with mutex algorithms, concurrent data structure algorithms, and thread abstraction in Chapter 4. Finally conclusions and future work are in Chapter 5.
Concurrent Programs

In this chapter, we formally describe the definition of concurrent programs in which each process can be expressed by an automaton. Then we define the TSO semantics by defining rewriting operations. Finally we show how to present executions of a concurrent program under SC and TSO semantics.

2.1 Preliminaries

We use \( \mathbb{N} \) to denote the set of natural numbers. For sets \( A \) and \( B \), we use \( f : A \to B \) to denote that \( f \) is a total function that maps \( A \) to \( B \), and use \( f : A \xrightarrow{B} B \) to denote that \( f \) is a partial function that maps \( A \) to \( B \).

Let \( A \) be a finite set. We use \( A^* \) and \( A^+ \) to denote the sets words and non-empty words respectively over \( A \); and use \( \epsilon \) to denote the empty word.

Consider a word \( w = a_1a_2 \cdots a_n \in A^* \). We define \( |w| := n \) and \text{last}(w) := a_n. For \( i : 1 \leq i < j \leq n \), we define \( w(i) := a_i \), \( w[i \cdots j] := a_ia_{i+1} \cdots a_j \) (i.e., it is the contiguous subword from position \( i \) to \( j \)), and \( w \ominus i := a_1 \cdots a_{i-1}a_{i+1} \cdots a_n \) (i.e., it is the result of deleting the \( i^{th} \) element from \( w \)). For \( a \in A \), we write \( a \in w \) if \( a \) appears in \( w \), i.e., \( a = w(i) \) for some \( i : 1 \leq i \leq |w| \).
We use $w_1 \cdot w_2$ to denote the concatenation of $w_1$ and $w_2$. 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 $B$; define $w \ominus B := i_{i_1} i_{i_2} \cdots i_{i_m}$, i.e., it gives the sequence of indices of the elements that belong to $B$; and define $w \oslash B$ to be the maximal subword of $w$ such that $a_{i_j} \notin B$ for all $j : 1 \leq j \leq m$, i.e., we remove the elements that belong to $B$.

2.2 Concurrent Programs

2.2.1 Process’s Automata

Fundamental knowledge of automata can be found in [16]. A program of a process $p$ is expressed by an automaton $A_p$. $A_p$ is a triple $\langle Q_p, \text{init}_p, \Delta_p \rangle$ where $Q_p$ is a finite set of process states, $\text{init}_p \in Q_p$ is the initial state, and $\Delta_p$ is a finite set of transitions. All final states of $p$ are contained in a set $F_p \subseteq Q_p$. In the initial state $\text{init}_p$, the program counter is at the entry of program, and all variables have value 0 as the initial value. Process communicates with each other by shared variables. Define $X$ as the set of all shared variables of all processes and $V$ as the finite domain of variables.

Each transition is a triple $\langle q, \text{op}, q' \rangle$ where $q, q' \in Q_p$ and $\text{op}$ is an operation. For a transition $t = \langle q, \text{op}, q' \rangle$, we define $\text{source} (t) := q$, $\text{target} (t) = q'$, and $[t] := \text{op}$. There are five operations:

- **No operation nop**: $p$ changes its program counter but keeps the same values of all variables.
- **Read operation read$(x, v)$**: $p$ reads a value $v$ from $x$. $\text{read}(x, v)$ is blocked if it is executed by $p$ and $p$ cannot read the value $v$ of $x$.
- **Write operation write$(x, v)$**: $p$ writes the value $v$ to a variable $x$.
- **Fence operation fence**: If we put fence between write and read, the ordering of
write before read is seen by every processes.

- **Atomic read-write operation** arw(x, v, v’): p atomically checks whether x has value v and if it does, p changes value of x to v’. Otherwise it writes value v to x.

where x ∈ X, and v, v’ ∈ V.

**Example 1:** For example, process p has its program code in Figure 1.3 and its automaton Ap = (Qp, initp, Δp) in Figure 2.1, where Qp = {q0, q1, q2, q3}, initp = q0 ∈ Qp, Δp = {t1, t2, t3} = {<q0, nop, q1>, <q1, write(x, 1), q2>, <q2, read(y, 0), q3>}.  

![Figure 2.1: Automaton of p.](image)

### 2.2.2 Concurrent Programs

A concurrent program P consists of a finite number of finite-state processes. Formally, a concurrent program is a pair P := (P, A) where P is a finite set of processes and A := {Ap | p ∈ P} is a set of finite-state automata (one automaton Ap for each process p ∈ P).

**Example 2:** For example, in Figure 1.3 program P has two processes P = {p, p’}, two automata Ap, Ap’ ∈ A, the set of X = {x, y}, and the domain V = {0, 1}.

We partition the set Δp into the subsets Δp^nop, Δp^write, Δp^read, Δp^arw, and Δp^fence, such that each subset contains the set of transitions performing the corresponding operation. For instance Δp^write contains all transitions t ∈ Δp where [t] is of the form write(x, v) for some x ∈ X and v ∈ V.

For i ∈ {nop, read, write, arw, fence}, we define Δi := ∪p∈P Δp^i. For a variable x ∈ X, we define Δp^write to be the subset of Δp^i operating on the variable x. For instance Δp^write contains all transitions t ∈ Δp where [t] is of the form write(x, v) for
some \( v \in V \). For a variable \( x \in X \) and value \( v \in V \), we define \( \Delta_p^{i,x,v} \) to be the subset of \( \Delta_p \) operating on the variable \( x \) and the value \( v \). For instance \( \Delta_p^{\text{write},x,v} \) contains all transitions \( t \in \Delta_p \) where \([t]\) is of the form \( \text{write}(x,v) \). Notice that \( \Delta_i^x = \cup_{v \in V} \Delta_p^{i,x,v} \), and that \( \Delta_p^{i,x} = \cup_{v \in V} \Delta_p^{i,x,v} \). We define \( \Delta^{i,x} := \cup_{p \in P} \Delta_p^{i,x} \). For instance \( \Delta_p^{\text{write},x} \) contains all transitions \( t \in \Delta_p \) where \([t]\) is of the form \( \text{write}(x) \). For \( i \in \{\text{read}, \text{write}, \text{arw}\} \), we use \( i(x) \) to denote an operation \( i \) to \( x \). For instance \( \text{write}(x) \) denote a write operation to \( x \). Moreover, we use \( \Delta_p^{\text{update},x,v} \) to denote a transition of \( p \) that changes the value of the variable \( x \) to the value \( v \) (including \text{write} and \text{arw} transitions), \( \Delta_p^{\text{update},x} \) to denote a transition that changes value the variable \( x \) (including \text{write} and \text{arw} transitions), and \( \text{update}(x,v) \) to denote an operation that changes value of the variable \( x \) to the value \( v \) (including \text{write} and \text{arw} operations). For a concurrent program \( P \), we define \( \Delta := \cup_{p \in P} \Delta_p \) (i.e, the set of all transitions of all processes).

2.3 Total Store Order Semantics

2.3.1 Sequential Paths

For a sequence of transitions \( \delta = t_1t_2 \cdots t_n \in \Delta^* \), we define \([\delta] := \text{op}_1\text{op}_2 \cdots \text{op}_n \) where \( \text{op}_i = [t_i] \) for \( i : 1 \leq i \leq n \) (i.e, the sequence of operations in a sequence of transitions). Consider a sequence of transitions of process \( p \): \( \sigma_p = t_1t_2 \cdots t_n \in \Delta^*_p \). We say that \( \sigma_p \) is a path of \( p \) if source \( (t_{i+1}) = \text{target} \ (t_i) \) for all \( i : 1 \leq i < n \). In the path \( \sigma_p \), \( p \) moves from \( q \) to \( q' : q \xrightarrow{t_1} q_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} q_n \) for \( q = \text{source} \ (t_1) \), \( q' = q_n \), \( q_0 = \text{target} \ (t_i) \) for all \( i : 1 \leq i \leq n \). We say that \( \sigma_p \) is a path from \( q \), and \( q = \text{start} \ (\sigma_p) \).

A path \( \sigma_p \) is a sequential path of process \( p \) if this sequence is a real path in the source code of \( p \). In other words, \( \sigma_p \) contains all operations of a path in the source code of \( p \) and the ordered of operations follows the program order. We use \( \Sigma_p^\delta \) to
denote the set of all sequential paths from \( q \) of \( p \), and \( \Sigma_p \) to denote the set of all sequential paths of \( p \): 
\[ \Sigma_p := \bigcup_{q \in Q_p} \Sigma_q^p. \]

**Example 3:** For example in Figure 2.1, \( p \) moves from \( q_0 \) to \( q_3 \): \( q_0 \xrightarrow{t_1} q_1 \xrightarrow{t_2} q_2 \xrightarrow{t_3} \) in a sequential path \( \sigma_p = t_1t_2t_3 \), \( [\sigma_p] = \text{nop}.\text{write}(x,1).\text{read}(y,0) \). A sequence of transitions \( \delta \) such that \( [\delta] = \text{nop}.\text{read}(y,0).\text{write}(x,1) \) is a path of \( p \) from \( q_0 \), but not a sequential path.

### 2.3.2 Total Store Order Rewriting

As saying in the Chapter 1, TSO model \([29, 32]\) allows reordering and read own write when executing programs. Given a path of process \( p \), \( \sigma_p \), we define *reordering* and *read-own-write* operations on \( \sigma_p \).

In a read own write, first \( p \) stores some write operations in its buffer, and the last write to \( x \) in the buffer is \( \text{write}(x,v) \). Then if a read operation \( \text{read}(x,v) \) is executed, it will read the value \( v \) of \( x \) from the buffer, not from the main memory. In this case, although the operation \( \text{write}(x,v) \) is not yet updated to the shared memory, an operation \( \text{read} \) to \( x \) of \( p \) can read the value \( v \). *Read-own-write operation* transforms a path \( \sigma_p \) to another path \( \sigma'_p \) such that they have the same starting state, and \( \sigma'_p \) contains the sequence of \( p \)'s operations seen from the view of memory, \( \sigma_p \overset{\text{row}}{\longrightarrow} \sigma'_p \):

- \( \text{start} (\sigma_p) = \text{start} (\sigma'_p) \).
- \( [\sigma_p] = \cdots \text{write}(x,v).(\text{nop})^*\text{read}(x,v)\cdots \overset{\text{row}}{\longrightarrow} [\sigma'_p] = \cdots (\text{nop})^*\text{write}(x,v)\cdots \).

In a reordering, first \( p \) stores some write operations in its buffer, but none of them writes to variable \( x \). Then if a read operation \( \text{read}(x,v) \) is executed, it will get value \( v \) of \( x \) from the main memory. *Reordering operation* transforms a path \( \sigma_p \) to another path \( \sigma'_p \) such that they have the same starting state, and \( \sigma'_p \) contains the sequence of \( p \)'s operations seen from the view of memory, \( \sigma_p \overset{\text{ro}}{\longrightarrow} \sigma'_p \):

14
• \( \text{start} (\sigma_p) = \text{start} (\sigma'_p) \).

• \( [\sigma_p] = \cdots . \text{write}(y, v').(\text{nop})^* . \text{read}(x, v) \cdots \llcorner_{ro} [\sigma_p]' = \cdots .(\text{nop})^* . \text{read}(x, v) . \text{write}(y, v') \cdots . \)

If we do not want to differ between reordering and read-own-write operations, we can call them with the same name as \emph{rewriting operation}, \( \llcorner_{ro} \).

2.3.3 Reordered Paths

We define \( \llcorner q_p \) (or \( \llcorner p \)) as the closure of \( \Sigma_q^p \) (or \( \Sigma_p \)) under rewriting operation:

\[
\Lambda_q^p = \{ \sigma'_p \in \Delta_p^* | \exists \sigma_p \in \Sigma_q^p : \sigma_p \llcorner_{ro} \sigma'_p \} \\
\Lambda_p = \{ \sigma'_p \in \Delta_p^* | \exists \sigma_p \in \Sigma_p : \sigma_p \llcorner_{ro} \sigma'_p \}
\]

We see that \( \Lambda_p := \bigcup_{q \in Q_p} \Lambda_q^p \).

We call each sequence \( \lambda_p \in \Lambda_p \) as a \emph{reordered path}. We see that a sequential path is also a reordered path without rewriting. From now, we use \( \sigma_p \) and \( \lambda_p \) for a sequential path and a reordered path respectively. If \( \sigma_p \llcorner_{ro} \lambda_p \), \( \sigma_p \) is called the \emph{original} of \( \lambda_p \). We define \emph{original} (\( \lambda_p \)) := \( \sigma_p \). In the case we can find more than one \( \sigma_p \) to be rewritten to \( \lambda_p \), \emph{original} (\( \lambda_p \)) will be the shortest sequential path. By this definition, \emph{original} (\( \lambda_p \)) is decided.

Example 4: In Figure 2.1 the sequential path \( \sigma_p = t_1t_2t_3 \), \( [\sigma_p] = \text{nop}.\text{write}(x, 1).\text{read}(y, 0) \), we have a reordered path \( \lambda_p \) such that \( \text{start} (\lambda_p) = \text{start} (\sigma_p) = g_0 \) and \( [\lambda_p] = \text{nop}.\text{read}(y, 0).\text{write}(x, 1) \). We have \emph{original} (\( \lambda_p \)) = \( \sigma_p \) because we can reorder \( \text{read}(y, 0) \) before \( \text{write}(x, 1) \) by applying the reordering operation on \( \sigma_p \).

2.4 Program Runs

2.4.1 Shuffling Operation

Shuffling operation \( \circ \) can be defined by an inductive way:

• \( \alpha \circ \epsilon = \alpha \).
• $\epsilon \circ \beta = \beta$.

• Otherwise $a.\alpha \circ b.\beta = a.(\alpha \circ b.\beta) \cup b.(a.\alpha \circ \beta)$.

where $a$ and $b$ are operations, $\alpha$ and $\beta$ are sequences of operations.

We use $T_{SC}(\mathcal{P})$ (resp. $T_{TSO}(\mathcal{P})$) to denote the set of all sequences of transitions created by applying the shuffling operation on sequential (resp. reordered) paths of processes (one path for each process):

- $T_{SC}(\mathcal{P}) = \{\sigma \in \Delta_p^* \mid [\sigma] = [\sigma_{p_1}] \circ [\sigma_{p_2}] \cdots \circ [\sigma_{p_n}], p_i \in \mathcal{P}, \sigma_{p_i} \in \Sigma_{p_i}, 1 \leq i \leq n\}$.
- $T_{TSO}(\mathcal{P}) = \{\lambda \in \Delta_p^* \mid [\lambda] = [\lambda_{p_1}] \circ [\lambda_{p_2}] \cdots \circ [\lambda_{p_n}], p_i \in \mathcal{P}, \lambda_{p_i} \in \Lambda_{p_i}, 1 \leq i \leq n\}$. 

2.4.2 Sequential Runs

A sequence of transitions $\pi = t_1 t_2 \cdots t_n \in \Delta^*$ is a sequential run if and only if (i) it is created by shuffling sequential paths of processes, i.e. $\pi \in T_{SC}(\mathcal{P})$; (ii) $\pi$ starts from the initial states of all processes; (iii) it does not contain any blocked read operation (see Section 2.2.1 for blocked read operation). The configuration in which all processes are in the initial states is the initial configuration $c_{init}$. Remember in $c_{init}$, all program counters are at the entry of program, and all variables have value 0. From $c_{init}$, a process can write directly to memory by write operations or read from memory by read operations. The set of all sequential runs is $\Pi^{SC}(\mathcal{P})$.

Example 5: In Figure 2.2, program $\mathcal{P}$ has two processes $P = \{p, p'\}$ with the source code of $p_1$ and $p_2$ as sequences $a_1$ and $b_1, b_2$ respectively. Then $\Pi^{SC}(\mathcal{P})$ can contain at most 9 sequential runs, including the empty word $\epsilon$. The exact number of sequential runs in $\Pi^{TSO}(\mathcal{P})$ depends on whether $p_1$ and $p_2$ contain any read operation or not.
2.4.3 Reordered Runs

A sequence of transitions $\pi = t_1 t_2 \cdots t_n \in \Delta^*$ is a reordered run if and only if (i) it is created by shuffling reordered paths of processes, $\pi \in T_{TSO}(\mathcal{P})$; (ii) $\pi$ starts from the initial states of all processes; and (iii) it does not contain any blocked read operation. As in sequential runs, the configuration in which all processes are in the initial states is the initial configuration $c_{init}$. From $c_{init}$, a process can write directly to memory by write operations or read from memory by read operations. The set of all reordered runs is $\Pi_{TSO}(\mathcal{P})$. 

Figure 2.2: Example of sequential runs.

Figure 2.3: Example of reordered runs.
Example 6: In Figure 2.3, following the above example, now $p'$ allows $b_2$ to be reordered before $b_1$. Then $\Pi^{\text{TSo}}(P)$ can contain at most 16 reordered runs, including the empty word $\epsilon$. The exact number of reordered runs in $\Pi^{\text{TSo}}(P)$ depends on whether $p_1$ and $p_2$ contain any read operation or not.
In this chapter, we define our persistence definition of concurrent programs, and then using this definition to construct a specific pattern for non-persist programs. With our pattern, we transform the persistence problem of programs under TSO to a reachability problem of programs under SC by a code-to-code translation. An fence insertion algorithm and compositional verification techniques are used to correct non-persist programs and reduce state explosion problem faced in programs with large number of processes.

3.1 Traces

3.1.1 Program Order

Recall that for a reordered path $\lambda_p$, $original(\lambda_p) = \sigma_p$ where $\sigma_p$ is transformed to $\lambda_p$ by the rewriting operation. Now for a run $\pi = t_1t_2\cdots t_n \in \Delta^*$ such that $[\pi] = [\lambda_{p_1}] \circ [\lambda_{p_2}] \cdots \circ [\lambda_{p_n}]$, we define $original(\pi) = t'_1t'_2\cdots t'_m \in \Delta^*$ such that $[original(\pi)] = [\sigma_{p_1}] \cdot [\sigma_{p_2}] \cdots [\sigma_{p_n}]$ where $original(\lambda_{p_i}) = \sigma_{p_i}$ for all $1 \leq i \leq n$. Notice that $original(\pi)$ can contain some bloked read operations, i.e. it can be that $original(\pi) \notin \Pi^{SC}(P)$. For a run $\pi$, we use $\pi_\sigma, \pi_p := \pi \odot \Delta_p$, to denote all transitions
of process $p$ in $\pi$.

Then we define the program order, $\text{ProgOrder} (\pi) : P \mapsto [\Delta^*]$, by $\text{ProgOrder} (\pi) (p) := [\text{original} (\pi) \odot \Delta_p] = [\text{original} (\pi_p)]$ for each $p \in P$. (Observe that an arw operation is considered as a write transition since it modifies the memory content.) In other words it extracts, for each process $p$, the sequence of nop, write, read, fence, and arw operations.

### 3.1.2 Total Store Order

We define the total store order by $\text{TSOrder} (\pi) := [\pi \odot (\Delta^{\text{write}} \cup \Delta^{\text{arw}})]$, i.e., it extracts the sequence of write and arw operations of all processes from $\pi$.

### 3.1.3 Traces

We define $\text{trace} (\pi) := \langle \text{ProgOrder} (\pi) , \text{TSOrder} (\pi) \rangle$. Below we define the persistence of concurrent programs based on traces.

### 3.1.4 Persistence

Consider a reordered run $\pi \in \Pi^{\text{TSO}} (P)$. Notice that by definition $\pi$ starts from the initial configuration $c_{\text{init}}$. We say that the run $\pi$ is persistent if there is a sequential run $\pi' \in \Pi^{\text{SC}} (P)$ such that $\text{trace} (\pi) = \text{trace} (\pi')$; otherwise we say that the run $\pi$ is fragile. We use $\Pi^{\text{Persist}} (P)$ to denote the set of all persistent reordered runs. The set of all reordered run that is fragile is $\Pi^{\text{Fragile}} (P)$. A concurrent program $P$ is said to be persistent if each run $\pi$ of $P$, $\pi \in \Pi^{\text{TSO}} (P)$, is persistent; otherwise it is called fragile. An instance of the persistence problem consists of a concurrent program $P$, and the question is whether $P$ is persistent or not.

### 3.1.5 Avoiding Bad Behaviors with Persistence

**Lemma 1.** If a concurrent program $P$ is persistent, $P$ does not contain non-SC behaviors under TSO.
**Proof.** By definition of persistence, for each reordered run $\pi \in \Pi^{\text{TSO}}(P)$, there exists a sequential run $\pi' \in \Pi^{\text{SC}}(P)$ such that $\text{trace}(\pi) = \text{trace}(\pi')$. Therefore, program orders of $\pi$ and $\pi'$ are the same, $\text{ProgOrder}(\pi) = \text{ProgOrder}(\pi')$; and total store orders of $\pi$ and $\pi'$ are also the same, $\text{TSOrder}(\pi) = \text{TSOrder}(\pi')$. Then we see that after running $\pi$ and $\pi'$, each process $p \in P$ has the same target state, and the target memories also are the same. It means that $\pi$ and $\pi'$ have the same behaviors. 

We use $\pi = \overline{\pi'}$ to indicate that 2 runs have the same target states of processes and the same target memories.

### 3.1.6 Minimal Fragility

A reordered run $\pi, \pi \in \Pi^{\text{TSO}}(P)$, is *minimally fragile* if $\pi$ is a shortest reordered run such that $\pi$ is fragile, i.e., (i) $\pi \in \Pi^{\text{Fragile}}(P)$, and (ii) all reordered runs $\pi' \in \Pi^{\text{TSO}}(P)$ with $|\text{original}(\pi')| < |\text{original}(\pi)|$ are persistent. We use $\Pi^{\text{MinFragile}}(P)$ to denote the set of reordered runs that are minimal fragile.

### 3.2 Solving Persistence

#### 3.2.1 Type $\oplus$ Runs

A reordered run $\pi, \pi \in \Pi^{\text{TSO}}(P)$, is of type $\oplus$ wrt. $p$ if the following three conditions are satisfied:

- $\pi \in \Pi^{\text{MinFragile}}(P)$, i.e., $\pi$ is minimally fragile.

- $\text{last}(\text{original}(\pi_p)) \in \Delta_p^{\text{read}}$, i.e., the last operation issued by $p$ in its program order is a read operation.

- $\text{last}(\text{original}(\pi_p)) \in \pi$, i.e., the last operation executed by $p$ in $\pi$ must appear in $\pi$. 

21
We use $\Pi_p^\downarrow (P)$ to denote the set of all reordered runs that are type $\downarrow$ wrt. $p$, and define $\Pi_p^{\downarrow} (P) := \bigcup_{p \in P} \Pi_p^\downarrow (P)$.

**Lemma 2.** $\Pi_p^{\downarrow} (P) = \Pi_{\textup{MinFragile}} (P)$.

**Proof.** The $\subseteq$-direction is trivial since, by definition, each run that is of that type $\downarrow$ wrt. any $p \in P$ is also minimally fragile. We show the $\supseteq$-direction by contradiction. Consider a run $\pi \in \Pi_{\textup{Fragile}} (P)$.

Suppose that there is no process $p \in P$ such that $\text{last} (\text{original} (\pi_p)) \in \Delta^{\text{read}}$. We have $\text{last} (\text{original} (\pi_p)) \in (\Delta^{\text{write}} \cup \Delta^{\text{nop}} \cup \Delta^{\text{fence}} \cup \Delta^{\text{arw}})$. Then $\text{last} (\pi) \in (\Delta^{\text{write}} \cup \Delta^{\text{nop}} \cup \Delta^{\text{fence}} \cup \Delta^{\text{arw}})$ and let $m = \text{last} (\pi \otimes (\Delta^{\text{write}} \cup \Delta^{\text{nop}} \cup \Delta^{\text{fence}} \cup \Delta^{\text{arw}}))$, i.e., $m$ is the index of the last operation. Define a new run $\pi_1$ by deleting the last transition $t_m$ from $\pi$, i.e., $\pi_1 := \pi \ominus m$. Observe that because $\pi_1 \in T_{\text{TSO}} (P)$ and $\pi_1$ is not blocked, $\pi_1 \in \Pi_{\text{TSO}} (P)$. Since $\pi$ by assumption is a shortest fragile run, it follows that $\pi_1$ is persistent and hence (by Lemma 1) there is a sequential run $\pi_2 \in \Pi_{\text{SC}} (P)$ such that $\text{trace} (\pi_2) = \text{trace} (\pi_1)$ and $\pi_2 = \pi_1$. Define $\pi_3 \in \Pi_{\text{SC}} (P)$ by $\pi_3 := \pi_2 \cdot \pi (m)$. It follows that $\text{trace} (\pi_3) = \text{trace} (\pi)$ and hence $\pi$ is persist which is a contradiction.

Now we prove that $\text{last} (\text{original} (\pi_p)) \in \pi$. Suppose that $\text{last} (\text{original} (\pi_p)) \notin \pi$, then $\text{last} (\text{original} (\pi_p)) \notin \pi_p$. Notice that $\text{last} (\text{original} (\pi_p)) \in \Delta^{\text{read}}$ (by (ii)). Let $k = |\text{original} (\pi_p)|$. Define a new sequential path $\sigma_p$ by deleting the last transition $t_k$ from $\text{original} (\pi_p)$, i.e., $\sigma_p := \text{original} (\pi_p) \ominus k$. We observe that $\sigma_p$ can be written to $\pi_p$ by the rewriting operation, and $|\sigma_p| < |\text{original} (\pi_p)|$. This means that $\sigma_p$ should be the original of $p$: $\sigma_p = \text{original} (\pi_p)$. This is infeasible because $\sigma_p \neq \text{original} (\pi_p)$. \hfill $\square$

### 3.2.2 Type $\circledast$ Runs

We use $\Pi_p^{\circledast} (P)$ to denote the set of reordered runs that are SC wrt. $p$. We say that $\pi$ is singly TSO wrt. $p \in P$, $\pi \in \Pi_p^{\text{SinglyTSO}} (P)$, if $\pi \in \Pi_p^{\circledast} (P)$ for all processes in $P \setminus \{p\}$,
i.e., \( \pi \) is SC wrt. all processes except (possibly) \( p \). A reordered run \( \pi, \pi \in \Pi^{TSO}(P) \), is of type \( \triangleright \) wrt. \( p \) if the following two conditions are satisfied:

- \( \pi \) is of type \( \triangleright \) wrt. \( p \).
- \( \pi \in \Pi_p^{SinglyTSO}(P) \).

We use \( \Pi_p^{\triangleright}(P) \) to denote the set of runs that are type \( \triangleright \) wrt. \( p \).

**Lemma 3.** \( \Pi_p^{\triangleright}(P) = \emptyset \) iff \( \Pi_p^{\triangleright}(P) = \emptyset \).

**Proof.** The only-if-direction holds trivially since, by definition, any reordered run that is of type \( \triangleright \) wrt. \( p \) is also of type \( \triangleright \) wrt. \( p \). We show the if direction. Consider a run \( \pi \in \Pi_p^{\triangleright}(P) \).

Let \( m \) be the maximal index of \( \pi \) such that \( [\pi(m)] = last(ProgOrder(\pi)(p)) \) and \( \pi(m) \in \Delta_p \), and define \( \pi_1 := \pi \ominus m \), i.e., we delete the last transition of \( \text{original}(\pi_p) \) from \( \pi \). Notice that this transition is a read transition according to Lemma 2. Since \( |\text{original}(\pi_1)| < |\text{original}(\pi)| \), we know that \( \pi_1 \in \Pi_p^{\text{Persist}}(P) \) and hence there is a \( \pi_2 \in \Pi_p^{\text{SC}}(P) \) such that \( \text{trace}(\pi_2) = \text{trace}(\pi_1) \). Define \( \pi_3 := \pi \ominus (\Delta_p \cup (\cup_{p' \in P \setminus \{p\}}(\Delta_{p'}^{\text{write}} \cup \Delta_{p'}^{\text{arw}}))) \). In other words, we derive \( \pi_3 \) from \( \pi \) by keeping all transitions of \( p \) and all write and arw transitions of other processes.

Notice that \( \pi_2 \ominus (\Delta_{\text{write}} \cup \Delta_{\text{arw}}) = \pi_3 \ominus (\Delta_{\text{write}} \cup \Delta_{\text{arw}}) \). Let \( \pi_2 \) be of the form \( \rho_0 \cdot t_1 \cdot \rho_1 \cdot t_2 \cdots \rho_{m-1} \cdot t_m \cdot \rho_m \) and let \( \pi_3 \) be of the form \( \rho'_0 \cdot t_1 \cdot \rho'_1 \cdot t_2 \cdots \rho'_{m-1} \cdot t_m \cdot \rho'_m \) such that \( \pi_2 \ominus (\Delta_{\text{write}} \cup \Delta_{\text{arw}}) = \pi_3 \ominus (\Delta_{\text{write}} \cup \Delta_{\text{arw}}) = t_1 t_2 \cdots t_m, t_i \in (\Delta_{\text{write}} \cup \Delta_{\text{arw}}) \) with \( 1 \leq i \leq m \). Define \( \rho''_i := \rho'_i \ominus (\rho_i \cup \rho'_{p \in P \setminus \{p\}} \Delta_{p'}) \), for \( i : 0 \leq i \leq m \). Define \( \pi_4 := \rho''_0 \cdot t_1 \cdot \rho''_1 \cdot t_2 \cdots \rho''_{m-1} \cdot t_m \cdot \rho''_m \). To prove that \( \pi_4 \in \Pi_p^{SinglyTSO}(P) \), we show that:

- \( \pi_4 \) is a reordered run.
- All \( p' \in P \setminus \{p\} \) are in SC.
• $\text{trace}(\pi_4) = \text{trace}(\pi)$.

Indeed, because of this construction, $\pi_4 \in \Pi^{TSO}(P)$, i.e., $\pi_4$ is a reordered run. Moreover, $\pi_4 \odot \Delta' = \pi_2 \odot \Delta'$ for all $p' \in P \setminus \{p\}$, so all $p' \in P \setminus \{p\}$ are in SC.

We observe that $\text{ProgOrder}(\pi_4)(p') = \text{ProgOrder}(\pi_2)(p') = \text{ProgOrder}(\pi_1)(p') = \text{ProgOrder}(\pi)(p')$ for all $p' \in P \setminus \{p\}$, $\text{ProgOrder}(\pi_4)(p) = \text{ProgOrder}(\pi_3)(p) = \text{ProgOrder}(\pi)(p)$, so $\text{ProgOrder}(\pi_4) = \text{ProgOrder}(\pi)$. Including with $\text{TSOrder}(\pi_4) = \text{TSOrder}(\pi_3) = t_1t_2\cdots t_m = \text{TSOrder}(\pi)$, we have $\text{trace}(\pi_4) = \text{trace}(\pi)$. 

3.2.3 Type $\ominus$ Runs

Recall that the meaning of notations $\Delta^\text{update}_{p,x,v}, \Delta^\text{update}_p$, and $\text{update}(x,v)$ are defined in the section about concurrent programs in Chapter 2. A reordered run $\pi \in \Pi^{TSO}(P)$ is of type $\ominus$ wrt. $p$ if the following four conditions are satisfied:

- $\pi$ is of type $\otimes$ wrt. $p$.
- $\text{last}(\pi \otimes \Delta^\text{read}_p) = \text{last}(\pi \otimes \Delta^\text{read}_p)$, i.e., the last read transition of $p$ reads the value $v$ from the variable $x$.
- $\text{last}(\pi \otimes \Delta^\text{read}_p) > \text{last}(\pi \otimes (\Delta^\text{write}_p \cup \Delta^\text{fence}_p \cup \Delta^\text{arw}_p))$, i.e., the process $p$ does not write to $x$ or make any fence or arw operation after its last read transition.
- $\text{last}(\pi \otimes \Delta^\text{update}_p) > \text{last}(\pi \otimes \Delta^\text{read}_p)$ for some $v' \neq v$ and $p' \neq p$, i.e., there is a process $p' \neq p$ that writes the value of $x$ to a value $v' \neq v$ after the last read transition of $p$.

We use $\Pi^\ominus_p(P)$ to denote the set of all reordered runs that are type $\ominus$ wrt. $p$.

Lemma 4. $\Pi^\ominus_p(P) = \Pi^\ominus_p(P)$.

24
Proof. The $\preceq$-direction holds trivially since, by definition, any run that is of type $\odot$ wrt. $p$ is also of type $\otimes$ wrt. $p$. We show the $\succeq$-direction. Consider a reordered run $\pi \in \Pi_p^\otimes (P)$.

Notice that $\text{last (original (} \pi_p)) = \text{last (} \pi \odot \Delta_p^{\text{read}} \text{)}$. Let $m = \text{last (} \pi \odot \Delta_p^{\text{read}} \text{)}$, and define $\pi_1 := \pi \odot m$, i.e., we delete the last transition of $\text{original (} \pi_p)$ from $\pi$. Since $|\text{original (} \pi_1)| < |\text{original (} \pi)|$, we know that $\pi_1 \in \Pi_{\text{Persist}}^\star (P)$, and hence there is a sequential run $\pi_2 \in \Pi_{\text{SC}}^\star (P)$ such that $\text{trace (} \pi_2) = \text{trace (} \pi_1)$. Define $\pi_3 := \pi_2 \cdot \text{last (} \pi \odot \Delta_p^{\text{read}} \text{)}$. In other words, we derive $\pi_3$ from $\pi_2$ by adding the last read transition of $p$ at the end of $\pi_2$. We observe that $\text{trace (} \pi_3) = \text{trace (} \pi)$, then because $\pi$ is fragile we have $\pi_3 \notin \Pi_{\text{TSO}}^\star (c)$ (i.e., $\pi_3$ is not a reordered run). But $\pi \in \Pi_{\text{TSO}}^\star (P)$, so $\text{last (} \pi \odot \Delta_p^{\text{read}} \text{)}$ in the form of $\text{read (} x, v \text{)}$ must read the value $v$ of $x$ from an $\text{update (} x, v \text{)}$ operation or from initial values; and there are some $\text{write (} x, v' \text{)}$ operations overwriting the value $v$ by $v'$ of $x$ in $\pi_3$. Because $\text{TSOrder (} \pi_3) = \text{TSOrder (} \pi)$, we have $\text{last (} \pi \odot \Delta_p^{\text{update,} x} \text{)} = \text{last (} \pi \odot \Delta_p^{\text{update,} x, v'} \text{)}$, i.e, the last write or atomic read-write operations to the variable $x$ changes value of $x$ to $v'$. In $\pi$, $\text{last (} \pi \odot \Delta_p^{\text{read}} \text{)}$ is not blocked, meaning that $\text{last (} \pi \odot \Delta_p^{\text{read}} \text{)}$ must be before $\text{last (} \pi \odot \Delta_p^{\text{update,} x, v'} \text{)}$. In other words, $\text{last (} \pi \odot \Delta_p^{\text{update,} x, v'} \text{)} > \text{last (} \pi \odot \Delta_p^{\text{read}} \text{)}$, i.e, in $\pi$ the last read operation $\text{read (} x, v \text{)}$ of $p$ appears before the last $\text{update (} x, v' \text{)}$ operation.

Then because of $\text{last (} \pi_p \text{)} = \text{last (} \pi \odot \Delta_p^{\text{read}} \text{)}$, $\text{last (} \pi \odot \Delta_p^{\text{update,} x, v'} \text{)} > \text{last (} \pi \odot \Delta_p^{\text{read}} \text{)}$, and TSO semantics we have $\text{last (} \pi \odot \Delta_p^{\text{update,} x, v'} \text{)} = \text{last (} \pi \odot \Delta_p^{\text{update,} x, v'} \text{)}$ for a process $p' \neq p$, i.e, the last $\text{update (} x, v' \text{)}$ operation in $\pi$ does not belong to $p$. Moreover, because of $\text{last (} \pi_p \text{)} = \text{last (} \pi \odot \Delta_p^{\text{read}} \text{)}$ and TSO semantics we have $\text{last (} \pi \odot \Delta_p^{\text{read}} \text{)} > \text{last (} \pi \odot (\Delta_p^{\text{write,} x} \cup \Delta_p^{\text{fence}} \cup \Delta_p^{\text{arw}}) \text{)}$. Otherwise, $\text{last (} \pi \odot \Delta_p^{\text{read}} \text{)}$ has been reordered over a transition $\Delta_p^{\text{write,} x} \cup \Delta_p^{\text{fence}} \cup \Delta_p^{\text{arw}}$ that is impossible in TSO semantics.

$\square$
3.2.4 Type \(\oplus\) Runs

A run \(\pi \in \Pi^{TSo}(P)\) is type \(\oplus\) wrt. \(p, x, v\) if the following five conditions are satisfied:

- \(\pi\) is of type \(\oplus\) wrt. \(p\).
- \(\text{last} \left( \pi \otimes (\Delta^\text{write} \cup \Delta^\text{arw}) \right) = \text{last} \left( \pi \otimes \Delta^\text{write}_p \right)\), i.e., the process \(p\) performs the last \text{write} transition among all the \text{write} and \text{arw} operations.
- \(\text{last} \left( \pi \otimes \Delta^\text{read}_p \right) < \text{last} \left( \pi \otimes \Delta^\text{write}_p \right)\), i.e., the process \(p\) performs at least one write transition after its last read transition.

We use \(\Pi^{\oplus}_{p,x,v}(P)\) to denote the set of all runs type \(\oplus\) wrt. \(p, x, v\), and define \(\Pi^{\oplus}_{p}(P) := \bigcup_{x \in X, v \in V} \Pi^{\oplus}_{p,x,v}(P)\).

**Lemma 5.** \(\Pi^{\oplus}_{p}(P) = \Pi^{\ominus}_{p}(P)\).

**Proof.** The \(\subseteq\)-direction holds trivially since, by definition, any run that is of type \(\oplus\) wrt. \(p\) is also of type \(\ominus\) wrt. \(p\). We show the \(\supseteq\)-direction. Consider a run \(\pi \in \Pi^{\ominus}_{p}(P)\).

Let \(p''\) be a process such that \(p'' \neq p\) and \(\text{last} \left( \bigcup_{p' \neq p} \Delta^\text{update}_{p',x,v'} \right) = \text{last} \left( \Delta^\text{update}_{p'',x,v'} \right)\) for some \(v' \neq v\), i.e., the last \text{update}(\(x, v'\)) of all processes, except \(p\), belongs to process \(p''\).

First we prove that \(\text{last} \left( \pi \otimes \Delta^\text{read}_p \right) < \text{last} \left( \pi \otimes \Delta^\text{write}_p \right)\) by showing that \(\text{last} \left( \pi \otimes \Delta^\text{read}_p \right) < \text{last} \left( \pi \otimes \Delta^\text{update}_{p',x,v'} \right) < \text{last} \left( \pi \otimes \Delta^\text{write}_p \right)\). Suppose that \(\text{last} \left( \pi \otimes \Delta^\text{update}_{p',x,v'} \right) > \text{last} \left( \pi \otimes \Delta^\text{write}_p \right)\). Define \(\pi_1\) be the tail part of \(\pi\) from the position \(\text{last} \left( \pi \otimes \Delta^\text{update}_{p',x,v'} \right)\) to the end of \(\pi\), and \(\pi_2\) be the head part of \(\pi\) from beginning to the position before \(\text{last} \left( \pi \otimes \Delta^\text{update}_{p',x,v'} \right)\). Formally \(\pi_1 := \pi[\text{last} \left( \pi \otimes \Delta^\text{update}_{p',x,v'} \right) \cdots]\), and \(\pi_2 = \pi \ominus \pi_1\). Because of \(|\text{original}(\pi_2)| < |\text{original}(\pi)|\), we have \(\pi_2 \in \Pi^{\text{Persist}}(P)\), and hence there is a sequential run \(\pi_3 \in \Pi^{\oplus}_{p,x,v}(P)\) such that \(\pi_2 \oplus \pi_3 \in \Pi^{\oplus}_{p,x,v}(P)\).
\( \Pi^{SC}(\mathcal{P}) \) such that \( \text{trace}(\pi_3) = \text{trace}(\pi_2) \). Next we define \( \pi_4 = \pi_3 \cdot \pi_1 \). We observe that in \( \pi_4 \) all processes are in SC and \( \text{trace}(\pi_4) = \text{trace}(\pi) \). It means that \( \pi \) is not fragile. This is a contradiction. Now we have proved that \( \text{last} \left( \pi \otimes \Delta_p^{\text{update},x,v'} \right) < \text{last} \left( \pi \otimes (\Delta_p^{\text{write}} \cup \Delta_p^{\text{arw}}) \right) \). Next because of \( \pi \in \Pi_p^3(\mathcal{P}) \), for some \( p' \neq p \) we have \( \text{last} \left( \pi \otimes \Delta_p^{\text{read}} \right) < \text{last} \left( \pi \otimes \Delta_p^{\text{update},x,v'} \right) \). Then by selecting \( p'' \) as above we also have \( \text{last} \left( \pi \otimes \Delta_p^{\text{read}} \right) < \text{last} \left( \pi \otimes (\Delta_p^{\text{write}} \cup \Delta_p^{\text{arw}}) \right) \).

Next we observe that \( \text{last} \left( \pi \otimes (\Delta_p^{\text{write}} \cup \Delta_p^{\text{arw}}) \right) = \text{last} \left( \pi \otimes \Delta_p^{\text{write}} \right) \). Otherwise \( \text{last} \left( \pi \otimes \Delta_p^{\text{read}} \right) \) has been reordered over a transition \( \Delta_p^{\text{arw}} \) that is impossible in TSO semantics. Finally we have \( \text{last} \left( \pi \otimes \Delta_p^{\text{read}} \right) < \text{last} \left( \pi \otimes \Delta_p^{\text{write}} \right) \) because \( \text{last} \left( \pi \otimes \Delta_p^{\text{read}} \right) < \text{last} \left( \pi \otimes (\Delta_p^{\text{write}} \cup \Delta_p^{\text{arw}}) \right) \) and \( \text{last} \left( \pi \otimes (\Delta_p^{\text{write}} \cup \Delta_p^{\text{arw}}) \right) = \text{last} \left( \pi \otimes \Delta_p^{\text{write}} \right) \).

Now we prove \( \text{last} \left( \pi \otimes (\Delta_p^{\text{write}} \cup \Delta_p^{\text{arw}}) \right) = \text{last} \left( \pi \otimes \Delta_p^{\text{write}} \right) \) by contradiction. Suppose that \( \text{last} \left( \pi \otimes (\Delta_p^{\text{write}} \cup \Delta_p^{\text{arw}}) \right) \neq \text{last} \left( \pi \otimes \Delta_p^{\text{write}} \right) \). It means that there is a process \( p' \neq p \) such that \( \text{last} \left( \pi \otimes \Delta_p^{\text{write}} \right) < \text{last} \left( \pi \otimes (\Delta_p^{\text{write}} \cup \Delta_p^{\text{arw}}) \right) \). Because \( \text{last} \left( \pi \otimes \Delta_p^{\text{read}} \right) < \text{last} \left( \pi \otimes \Delta_p^{\text{write}} \right) \), we can delete the tail part of \( \pi \) from the last write or atomic read-write of process \( p' \) to the end, and the remaining of \( \pi \) we keep is a reordered run. Let \( \pi_5 \) and \( \pi_6 \) be the remaining and the tail part above. Because \( |\pi_5| < |\pi| \), we have a sequential run \( \pi_5 \) that has the same trace as \( \pi_5 \). Concatenate \( \pi_7 \) and \( \pi_6 \), we have \( \pi_8: \pi_8 = \pi_7 \cdot \pi_6 \). We observe that \( \pi_8 \) is a sequential run that has the same trace as \( \pi \). It means that \( \pi \) is persistent. That is a contradiction. \( \square \)

### 3.2.5 Type \( \odot \) Runs

A run \( \pi \in \Pi^{TSO}(\mathcal{P}) \) is type \( \odot \) wrt. \( p, x, v \) if there are runs \( \pi_1, \pi_2, \pi_3, \pi_4 \) such that the following five conditions are satisfied:

- \( \pi = \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4 \).
- \( \pi_1 \) is SC.
\( \pi_2 \in (\Delta_p^{\text{read}} \cup \Delta_p^{\text{nop}})^* \), and \( \text{last}(\pi_2) \in \Delta_p^{\text{read},x,v} \).

\( \pi_3 \in \left( (\cup_{p' \in P-p} \Delta_{p'}) \cup (\cup_{x' \in X-\{x\}} \Delta_{p'}^{\text{write},x',v} \cup \Delta_{p'}^{\text{nop}}) \right)^* \), and \( \text{last}(\pi_3) \in \Delta_{p'}^{\text{write},x,v} \) for some \( p' \neq p \) and \( v' \neq v \).

\( \pi_4 \circ \left( \cup_{p' \in P} \Delta_{p'}^{\text{write},x} \right) = \epsilon \), and \( \text{last}(\pi_4) \in \Delta_{p'}^{\text{write}} \).

We use \( \Pi^{\circ}_p(x,v)(P) \) to denote the set of runs type \( \circ \) wrt. \( p \), \( x,v \); define \( \Pi^{\circ}_p(x,v)(P) := \cup_{x \in X,v \in V} \Pi^{\circ}_{p,x}(P) \), and define \( \Pi^{\circ}_p(P) := \cup_{p \in P} \Pi^{\circ}_p(P) \).

**Lemma 6.** \( \Pi^{\circ}_p(P) = \emptyset \) implies \( \Pi^{\circ}_p(P) = \emptyset \).

**Proof.** Consider a run \( \pi \in \Pi^{\circ}_p(P) \). Let \( m = \text{last}(\pi \circ \Delta_p^{\text{read}}) \), i.e., \( m \) is the index of last read operation of \( p \) in \( \pi \). Then define \( t_{p,\text{after_closet}}(\text{last}(\pi \circ \Delta_p^{\text{read}})) \) to be the closet write transition of \( p \) after last \( \pi \circ \Delta_p^{\text{read}} \) in \( \pi \). Delete from \( \pi \) all transitions \( t \in \Delta_{p'} \) after last \( \pi \circ \Delta_p^{\text{read}} \) in \( \pi \), we have a run \( \pi_1 \). Then because of \( \pi \in \Pi^{\circ}_p(P) \), so by Lemma 4 \( \text{last}(\pi \circ \Delta_{p'}^{\text{update},x,v'}) > \text{last}(\pi \circ \Delta_p^{\text{read}}) \) for some \( v' \neq v \) and \( p' \neq p \). It means that we have deleted from \( \pi \) at least the transition \( \text{last}(\pi \circ \Delta_{p'}^{\text{update},x,v'}) \). Moreover, because \( \pi[m + 1 \cdots] \) does not have any read of \( p \), \( \pi_1 \) is a reordered run. Hence from \( \vert \pi \vert > \vert \pi_1 \vert \), we have \( \pi_1 \in \Pi^{\text{Persist}}(P) \), and there is a sequential run \( \pi_2 \in \Pi^{\text{Persist}}(P) \) such that \( \text{trace}(\pi_2) = \text{trace}(\pi_1) \). In \( \pi_2[t_{p,\text{after_closet}}(\text{last}(\pi \circ \Delta_p^{\text{read}})) \cdots] \), we delete all read operations of \( p \) that get value from some write operations, write \( \in \pi_2[t_{p,\text{after_closet}}(\text{last}(\pi \circ \Delta_p^{\text{read}})) \cdots] \), and reorder the remaining read operations just before the position of \( t_{p,\text{after_closet}}(\text{last}(\pi \circ \Delta_p^{\text{read}})) \) in \( \pi_2 \). We can do this by applying the rewriting operation on \( (\pi_2)_p \) to get \( (\pi_3)_p \), and then shuffling \( (\pi_3)_p \) with other \( (\pi_2)_p \). At the end of this step, from \( \pi_2 \) we have \( \pi_3 \) that \( \pi_3 \circ \Delta_p \) has the form \( \rho_0 \cdot (\mu_{\text{update}})_{1} \cdot \rho_1 \cdot (\mu_{\text{update}})_{2} \cdot \cdots (\mu_{\text{update}})_{j} \cdot \rho_j \cdot (\Delta_p^{\text{read}} \cup \Delta_p^{\text{nop}}) \cdot \text{last}(\pi \circ \Delta_p^{\text{read}}) \) \( * \cdot \text{last}(\pi \circ \Delta_p^{\text{read}}) \cdot t_{p,\text{after_closet}}(\text{last}(\pi \circ \Delta_p^{\text{read}})) \cdot \rho_{j+1} \cdot \)
\[(\mathsf{t}_{\text{update}})_j \cdots (\mathsf{t}_{\text{update}})_{m-1} \cdot (\mathsf{t}_{\text{update}})^m \cdot \rho_m.\] Here we use \(\mathsf{t}_{\text{update}}\) to denote a write or atomic read-write operation, and \(\mathsf{t}_{\text{update}}\) denote a write or atomic read-write of process \(p\).

Let \(\pi\) be of the form \(\rho'_0 \cdot (\mathsf{t}_{\text{update}})_1 \cdot \rho'_1 \cdot (\mathsf{t}_{\text{update}})_2 \cdots (\mathsf{t}_{\text{update}})_j \cdot \rho'_j \cdot \mathsf{last}(\pi \odot \Delta_{\text{read}}^p) \cdot \rho' \cdot t_{\text{after\_close}}(\mathsf{last}(\pi \odot \Delta_{\text{read}}^p) \cdot \rho_{j+1} \cdot (\mathsf{t}_{\text{update}})_j \cdots (\mathsf{t}_{\text{update}})) \cdot \rho_m \cdot (\mathsf{t}_{\text{update}})_j \cdot (\mathsf{t}_{\text{update}})_{m-1} \cdot (\mathsf{t}_{\text{update}})^m \cdot \rho'_m.\) Define

\[\rho''_i := \rho_i \cdot (\rho' \circ \odot \rho_{j+1} \circ \rho_m \circ (\mathsf{t}_{\text{update}})) \ast \mathsf{last}(\pi \odot \Delta_{\text{read}}^p) \cdot \rho' \quad \text{and} \quad \pi_4 := \rho''_0 \cdot (\mathsf{t}_{\text{update}})_1 \cdot \rho''_i \cdot (\mathsf{t}_{\text{update}})_{j} \cdot (\mathsf{t}_{\text{update}})_{m-1} \cdot (\mathsf{t}_{\text{update}})^m.\]

Finally, we delete from \(\pi_4\) all operations of \(p' \neq p\) after \(\mathsf{last}(\pi \odot \Delta_{\text{update}}^p, x, v')\), and have \(\pi_5\) such that \(\pi_5 \in \Pi_{\text{fragile}}(P)\).

\[\square\]

### 3.2.6 Combination

**Lemma 7.** \(\Pi_{\text{fragile}}(P) \neq \emptyset\) implies \(\Pi_{\text{fragile}}(P) \neq \emptyset\).

**Proof.** Using Lemma 2, Lemma 3, Lemma 4, Lemma 5, and Lemma 6 we have \(\Pi_{\text{fragile}}(P) \neq \emptyset\) implies \(\Pi_{\text{fragile}}(P) \neq \emptyset\). Then by the definition of minimal fragility, \(\Pi_{\text{fragile}}(P) \neq \emptyset\) implies \(\Pi_{\text{fragile}}(P) \neq \emptyset\). Finally we have \(\Pi_{\text{fragile}}(P) \neq \emptyset\) implies \(\Pi_{\text{fragile}}(P) \neq \emptyset\).

\[\square\]

From the previous lemmas we get the following theorem.

**Theorem 1.** \(\Pi_{\text{fragile}}(P) = \emptyset\) iff \(\Pi_{\text{fragile}}(P) = \emptyset\).

For \(i \in \{1, 2, 3, 4\}\) we define \(\Pi_i(\mathcal{P}) := \bigcup_{p \in P} \Pi_i(P)\), and define \(\Pi(\mathcal{P}) := \Pi(\mathcal{P})\).

**Proof.** The if-direction holds by Lemma 7. We proof the only-if-direction. Suppose that a run \(\pi \in \Pi_{\text{fragile}}(P)\). Then \(\pi\) is reordered run. But we cannot find any sequential run that has the same program order and total store order as in \(\pi\). The reason
is in \( \pi \), the last read operation of \( p \), \( \text{last} (\pi \odot \Delta^\text{read}_p) \), is after the last write operation, \( \text{last} (\pi \odot \Delta^\text{write}_p) \), in its program order; but appears before \( \text{last} (\pi \odot \Delta^\text{write}_p) \) when executing. So \( \pi \) is fragile, and \( \mathcal{P} \) is not persistent.

\( \square \)

### 3.3 Code-to-Code Translation

We translate the persistence problem for finite-state programs running under the TSO to the the reachability problem for finite-state program running under the SC semantics. We exploit Theorem 1 which shows that the persistence problem is reducible to the problem of checking whether the program has a type \( \oplus \) computation. Given a program \( \mathcal{P} \) we derive a new program \( \mathcal{P}^\text{SC} \) and translate the problem of whether \( \mathcal{P} \) has a run of type \( \oplus \) from its initial configuration to an instance the reachability problem defined on \( \mathcal{P}^\text{SC} \). Intuitively, each process \( p \) in \( \mathcal{P}^\text{SC} \) will simulate the corresponding process in \( p \) in \( \mathcal{P} \). Each run of \( \mathcal{P}^\text{SC} \) is divided into four phase: 0, 1, 2, 3. Each phase will accomplish a particular task in the simulation of \( \mathcal{P}^\text{SC} \).

Recall that a run of type \( \oplus \) is also of type \( \ominus \) and hence it is singly TSO, i.e., one process will be perform TSO transitions, while the rest will behave like SC processes. In phase 0 one process will (nondeterministically) be chosen to play the role of the TSO process. We refer to this process as the \( A \)-process, while we refer the other processes as the \( B \)-processes. All processes (including the \( A \)-process) will exhibit SC behaviors (recall that we are only considering SC runs of \( A^\text{SC} \)). In phase 1 all the processes will memic the SC behaviors of their counter-parts in \( \mathcal{P} \). In addition, any write operation, performed on a variable \( x \), will also be performed on a copy of \( x \) which we call \( x^\text{local} \). Thus, at the end of phase 1 each \( x^\text{local} \) will have the same value as \( x \). In phase 2 process \( A \) reads from the local memory, and writes both local and main memories. Processes \( B \) access to main memory. In phase 3 processes \( B \) will be halted, while process \( A \) is still in progress and checks for each its read operation.
whether $A$ can find a violation or not.

Suppose that we are given a concurrent program $P = \langle P, A \rangle$, with $A_P = \langle Q_p, init_p, \Delta_p \rangle$ operating on a set $X$ of shared variables. We will derive a new program $P_{SC} = \langle P_{SC}, A_{SC} \rangle$, with $A_{SC} = \{ A_p^{SC} | p \in P \}$ and $A_{p}^{SC} = \langle Q_p^{SC}, init_p^{SC}, \Delta_p^{SC} \rangle$, operating on an extended set $X^{SC}$ of shared variables. In our description of $P_{SC}$, we will write "$\Delta_p^{SC}$ contains $\langle q, op_1, op_2, \ldots, op_n, q' \rangle$" to describe that $\Delta_p^{SC}$ contains a set of transitions $\langle q, op_1, \text{tmp}_1 \rangle, \langle \text{tmp}_1, op_2, \text{tmp}_2 \rangle, \ldots, \langle \text{tmp}_{n-1}, op_n, q' \rangle$, where $\text{tmp}_1, \text{tmp}_2, \ldots, \text{tmp}_{n-1}$ are unique temporary states. For an operation performed on a variable $x$, we use $op[x/y]$ to denote the operation we get by replacing the variable $x$ by $y$. For instance if $op$ is of the form $\text{write}(x, v)$ then $op[x/y]$ denotes the operation $\text{write}(y, v)$.

Below, we describe how we construct $P_{SC}$ by first defining the set of shared variables on which it operates, and then introducing the set of states and the set of transitions for each process.

Variables. The set $X^{SC}$ contains all the variables in $X$. In addition, it contains the following extra variables:

- For each variable $x \in X$, the set $X^{SC}$ contains an additional copy $x^{\text{local}}$ of $x$. This variable will eventually only be updated by the TSO process.

- The variable $\text{ph}$ with domain $\{-1, 0, 1, 2, 3\}$. A run of $P_{SC}$ will consist of four phases. The current value of $\text{ph}$ gives the current phase. $-1$ is a special value for phase.

- The variable $\text{selected}$ with domain $\{0, 1\}$ indicates whether the TSO process has been selected or not.

- The variable $x^{\text{modified}}$ with domain $\{0, 1\}$ indicates whether variable $x$ has been changed in phase 2 or phase 3, or not.
• The state $q_{\text{fail}}$ denotes a failure state, meaning that we don’t find a violation. In other hand, the state $q_{\text{suc}}$ indicates that we have found a violation.

**States.** Each process $p$ has a new initial state $\text{init}^SC_p$. Furthermore, for each state $q \in Q_p$, the set $Q^SC_p$ contains the two copies $q_A$ and $q_B$. The copy $q_A$ is used if the process is acting as the $A$-process during the current run, while the copy $q_B$ is used if the process is a $B$-process.

**Phase 0.** The set $\Delta^SC_p$ contain the following two transitions:

- $\langle \text{init}^SC_p, \text{arw}(\text{selected}, 0, 1), (\text{init}_p)_A \rangle$. A process $p$ may declare itself being the $A$-process, and moves to the $a$-copy of its initial state. At the same time it sets the flag $\text{selected}$ to 1 thus preventing all other processes from acting as the $A$-process.

- $\langle \text{init}^SC_p, \text{nop}, (\text{init}_p)_B \rangle$. A process $p$ may decide to be a $B$-process by moving to the $B$-copy of its initial state.

**Phase 1.**

- For each state $q \in Q_p$, the set $\Delta^SC_p$ contains the transition $\langle q_A, \text{read}(\text{ph}, 0), \text{arw}(\text{ph}, 0, 1), q_A \rangle$.

- For each transition $\langle q, op, q' \rangle \in \Delta_p - \Delta^{\text{write}}_p - \Delta^{\text{arw}}_p$, the set $\Delta^SC_p$ contains the two transitions:
  - $\langle q_A, \text{read}(\text{ph}, 1), \text{arw}(\text{ph}, 1, -1), op, \text{write}(\text{ph}, 1), q'_A \rangle$.
  - $\langle q_B, \text{read}(\text{ph}, 1), \text{arw}(\text{ph}, 1, -1), op, \text{write}(\text{ph}, 1), q'_B \rangle$.

Regardless of whether $p$ is an $A$- or $B$-process, a non-write transition will be emulated. Notice that, each transition will first check the phase (here whether it is equal to 1), and disables the other processes (by atomically setting the
phase to \(-1\). After the operation \(op\) is performed, the phase is set back to 1. The states are changed (from \(q\) to \(q'\)) following the corresponding transition in \(\mathcal{P}\).

- For each transition \(\langle q, \text{write}(x, v), q' \rangle \in \Delta^\text{write}_p\), the set \(\Delta^\text{SC}_p\) contains the two transitions:
  
  \[ \langle q_A, \text{read}(ph, 1), \text{arw}(ph, 1, -1), \text{write}(x, v), \text{write}(x^{\text{local}}, v), \text{write}(ph, 1), q'_A \rangle. \]
  
  \[ \langle q_B, \text{read}(ph, 1), \text{arw}(ph, 1, -1), \text{write}(x, v), \text{write}(x^{\text{local}}, v), \text{write}(ph, 1), q'_B \rangle. \]

A write transition in \(\mathcal{P}\) is simulated by two transitions in \(\mathcal{P}^{\text{SC}}\), namely one which writes to the same variable \(x\) and one which performs the same operation on the copy \(x^{\text{local}}\).

- For each transition \(\langle q, \text{arw}(x, v, v'), q' \rangle \in \Delta^\text{arw}_p\), the set \(\Delta^\text{SC}_p\) contains the two transitions:
  
  \[ \langle q_A, \text{read}(ph, 1), \text{arw}(ph, 1, -1), \text{arw}(x, v, v'), \text{arw}(x^{\text{local}}, v, v'), \text{write}(ph, 1), q'_A \rangle. \]
  
  \[ \langle q_B, \text{read}(ph, 1), \text{arw}(ph, 1, -1), \text{arw}(x, v, v'), \text{arw}(x^{\text{local}}, v, v'), \text{write}(ph, 1), q'_B \rangle. \]

A \text{arw} transition in \(\mathcal{P}\) is simulated by two transitions in \(\mathcal{P}^{\text{SC}}\), namely one which atomic read-writes to the same variable \(x\) and one which performs the same operation on the copy \(x^{\text{local}}\).

\textbf{Phase 2.}

- For each transition \(\langle q, \text{write}(x, v), q' \rangle \in \Delta^\text{write}_p\), the set \(\Delta^\text{SC}_p\) contains the transition:
\[
\langle q_A, \text{read(ph, 1)}, \text{arw(ph, 1, 2)}, \text{write}(x, v), \text{write}(x_{\text{local}}, v), \text{write}(x_{\text{modified}}, 1), q_A' \rangle.
\]

• For each transition \( \langle q, \text{write}(x, v), q' \rangle \in \Delta_p^{\text{write}} \), the set \( \Delta_p^{\text{SC}} \) contains the transition:

\[
\langle q_A, \text{read(ph, 2)}, \text{arw(ph, 2, -1)}, \text{write}(x, v), \text{write}(x_{\text{local}}, v), \text{write}(x_{\text{modified}}, 1) \text{ write(ph, 2), } q_A' \rangle.
\]

• For each transition \( \langle q, \text{fence}, q' \rangle \in \Delta_p^{\text{write}} \), the set \( \Delta_p^{\text{SC}} \) contains the transition:

\[
\langle q_A, \text{read(ph, 2)}, \text{arw(ph, 2, -1), } q_{\text{fail}} \rangle.
\]

• For each transition \( \langle q, \text{arw}(x, v, v') \rangle \in \Delta_p^{\text{write}} \), the set \( \Delta_p^{\text{SC}} \) contains the transition:

\[
\langle q_A, \text{read(ph, 2)}, \text{arw(ph, 2, -1), } q_{\text{fail}} \rangle.
\]

• For each transition \( \langle q, \text{op}, q' \rangle \in \Delta_p - \Delta_p^{\text{write}} - \Delta_p^{\text{fence}} - \Delta_p^{\text{arw}} \), the set \( \Delta_p^{\text{SC}} \) contains the transition:

\[
\langle q_A, \text{read(ph, 2)}, \text{arw(ph, 2, -1), op}[x/x_{\text{local}}], \text{write(ph, 2), } q_A' \rangle.
\]

• For each transition \( \langle q, \text{op}, q' \rangle \in \Delta_p \), the set \( \Delta_p^{\text{SC}} \) contains the transition:

\[
\langle q_B, \text{read(ph, 2)}, \text{arw(ph, 2, -1), op, write(ph, 2), } q_B' \rangle.
\]

Phase 3.

• For each transition \( \langle q, \text{write}(x, v), q' \rangle \in \Delta_p^{\text{write}} \), the set \( \Delta_p^{\text{SC}} \) contains the transition:

\[
\langle q_A, \text{read(ph, 2)}, \text{arw(ph, 2, 3)}, \text{write}(x, v), \text{write}(x_{\text{local}}, v), \text{write}(x_{\text{modified}}, 1), q_A' \rangle.
\]
• For each transition \( \langle g, \text{read}(x, v), q' \rangle \in \Delta_p^{\text{read}} \), the set \( \Delta_p^{SC} \) contains the following transition
  \[ \langle g_a, \text{read}(\text{ph}, 3), \text{read}(x, v), \text{read}(x_{\text{local}}, v), q'_a \rangle. \]

• For each transition \( \langle g, \text{read}(x, v), q' \rangle \in \Delta_p^{\text{write}} \), the set \( \Delta_p^{SC} \) contains the following transitions, for each \( v' \neq v \):
  \[ \langle g_a, \text{read}(\text{ph}, 3), \text{read}(x, v'), \text{read}(x_{\text{local}}, v'), \text{read}(x_{\text{modified}}, 0), q'_{\text{suc}} \rangle. \]

• For each transition \( \langle g, \text{arw}(x, v, v') \rangle \in \Delta_p^{\text{write}} \), the set \( \Delta_p^{SC} \) contains the transition:
  \[ \langle g_a, \text{read}(\text{ph}, 3), \text{arw}(\text{ph}, 3, -1), q'_{\text{fail}} \rangle. \]

• For each transition \( \langle g, \text{fence}, q' \rangle \in \Delta_p^{\text{write}} \), the set \( \Delta_p^{SC} \) contains the transition:
  \[ \langle g_a, \text{read}(\text{ph}, 3), \text{arw}(\text{ph}, 3, -1), q'_{\text{fail}} \rangle. \]

**Theorem 2.** Program \( P^{SC} \) reaches to the state \( q_{\text{suc}} \) if and only if \( \Pi_p^{\oplus}(P) \neq \emptyset \).

Since the reachability problem is \textsc{Pspace-complete} for finite-state programs under SC, we obtain the following result as an immediate corollary of Theorem 1 and Theorem 2.

**Theorem 3.** The persistence problem for finite-state programs is \textsc{Pspace-complete}.

### 3.4 Fence Insertion

#### 3.4.1 Dangerous Attacks

For a concurrent program \( P \), one process will be selected as \( A \) process, other processes will be \( B \) processes. We try all attacks in from of \( \langle A, \text{write}_A, \text{read}_A \rangle \). \text{write}_A will be the first stored write operations of \( A \) in its buffer. We check whether \( \text{read}_A \) can be
reordered over write$_A$, and with operations of $B$ processes it forms a run $\pi \in \Pi^{(5)}(\mathcal{P})$. If it forms a violation, we say that this attack is dangerous, otherwise it is called safe.

Define $S := \{ \langle A, \text{write}_A, \text{read}_A \rangle | A \in \mathcal{P}, \text{write}_A \in \Delta^\text{write}_A, \text{read}_A \in \Delta^\text{read}_A \}$, i.e., $S$ is the set of all attacks of $\mathcal{P}$, and let $D \subseteq S$ be the set of all dangerous attacks.

### 3.4.2 Dangerous Paths

For a dangerous attack $\langle A, \text{write}_A, \text{read}_A \rangle \in D$, we have many paths in control flow graph of $A$ from write$_A$ to read$_A$. If we put at least one fence in each path, this attack will be safe. A path is called safe if we do not need to insert fence in this path to keep the attack safe, otherwise it is called dangerous. We find a dangerous path $pth$ by putting one fence in each path from write$_A$ to read$_A$, except the path $pth$. If the attack in this case is dangerous, the path $pth$ is dangerous.

Let $PTH(\langle A, \text{write}_A, \text{read}_A \rangle)$ be the set of all paths from write$_A$ to read$_A$ in a dangerous attack $\langle A, \text{write}_A, \text{read}_A \rangle$. For a set of paths, $Pths$, let $\text{Fence}(Pths)$ be the set of fences by putting each fence in each path of $Pths$. Let $DanPths$ be the set of all dangerous paths.

### 3.4.3 Algorithm

We present our dangerous-path detection algorithm (Algorithm 1). It inputs a concurrent program $\mathcal{P}$ and returns the set $DanPths$ containing all dangerous paths of $\mathcal{P}$.

First the algorithm finds all dangerous attacks in the set $S$ of $\mathcal{P}$, and stores them to the set $D$. Next for each dangerous attack in $D$, the algorithm finds all dangerous paths. At the end, the set of all dangerous paths of $\mathcal{P}$, $DanPths$, is returned.

**Theorem 4.** Algorithm 1 terminates and returns the set of all dangerous paths of $\mathcal{P}$.
Algorithm 1: Dangerous-path detection.

**input**: A concurrent program $P$  
**output**: The set of all dangerous paths of $P$, $DanPths$.

1. $DanPths \leftarrow \{\emptyset\}$, $D \leftarrow \{\emptyset\}$;
2. Calculate $S$ of $P$;
3. foreach $\langle A, \text{write}_A, \text{read}_A \rangle \in S$ do
   4. if $\langle A, \text{write}_A, \text{read}_A \rangle$ is dangerous then
      5. $D = D \cup \{\langle A, \text{write}_A, \text{read}_A \rangle\}$;
4. foreach $\langle A, \text{write}_A, \text{read}_A \rangle \in D$ do
   5. foreach $pth \in PTH(\langle A, \text{write}_A, \text{read}_A \rangle)$ do
      6. if $P$ w.r.p Fence($Pths$) is not persistent then
         7. $DanPths = DanPths \cup \{pth\}$;
6. return $DanPths$;

3.4.4 Integer Linear Programming

Let $pth_1, pth_2 \ldots, pth_n$ be dangerous paths of program $P$. Then let $t_1, t_2 \ldots, t_m$ be transitions in dangerous paths. We define a matrix $\text{Belong}[n][m]$ with $n$ rows and $m$ columns such that for all $1 \leq i \leq n, 1 \leq j \leq m$:

$$ \text{Belong}[i][j] = \begin{cases} 1, & t_j \in \text{att}_i \\ 0, & \text{otherwise} \end{cases}. $$

$\text{Belong}[i][j]$ is set to 1 if the transition $t_j$ is on the path $pth_i$, otherwise $\text{Belong}[i][j]$ is 0.

To make a program safe, for each dangerous path, we need to put at least one fence in the path from $\text{write}_A$ to $\text{read}_A$. A fence set is a minimal fence set of program $P$ if (i) $P$ is safe according to this fence set, and (ii) the number of fences in the set is minimal. Now we find a minimal fence set for $P$ by a integer linear programming:

$$ \begin{align*} \text{Min} : f(x) &= x_1 + x_2 + \cdots + x_m. \\
\text{Belong}[1][1] \cdot x_1 + \text{Belong}[1][2] \cdot x_2 + \cdots + \text{Belong}[1][m] \cdot x_m &\geq 1. \\
\cdots \\
\text{Belong}[n][1] \cdot x_1 + \text{Belong}[n][2] \cdot x_2 + \cdots + \text{Belong}[n][m] \cdot x_m &\geq 1. \\
x_1, x_2, \ldots, x_m &\text{ are binary.} \end{align*} $$
We see that the condition $\text{Belong}[i][1] \times x_1 + \text{Belong}[i][2] \times x_2 + \cdots + \text{Belong}[i][m] \times x_m \geq 1$ guarantees that we insert at least one fence in a dangerous path $p_{th_i}$. The condition $\text{Min}: f(x) = x_1 + x_2 + \cdots + x_m$ indicates that the number of fences is minimal.

3.5 Compositional Verification

Let $P$ be a concurrent program. In the following, we develop a compositional analysis for deciding persistence for programs under TSO in which we consider each process separately along with an abstraction of the other processes (i.e., environment processes). Observe that all the environment processes are supposed to be executed under SC. 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 transitions or the changes performed by the other processes over the set variables $X_p$ that $p$ can read from.

Then, we construct the environment processes such that the process $p$ has at least the same set memory views as in $P$. This ensures that the persistence of the constructed abstract program (i.e., $p$ along with the set of environment processes) wrt. $p$ implies the persistence of the original program $P$ wrt. $p$.

The rest of this section is organized as follows: First we introduce some definition and notation to formally define the memory view of a process. We then give some properties that the environment processes should satisfy and show that if these properties are satisfied then the persistence of the abstract program wrt. $p$ implies the persistence of the original program $P$ wrt. $p$. Finally, we give two instances of possible environment processes.

3.5.1 Definitions and Notations

For every process $p \in P$, we define $X_p \subseteq X$ to be the subset of variables to which $p$ has a read operations. Formally, $X_p = \{x \mid \Delta_{p,\text{read},x} \cup \Delta_{p,\text{write},x} \neq \emptyset\}$. Let $[\Delta_{p,\text{write},x}] = \{[t] \mid t \in \Delta_{p,\text{write},x} \}$. 

38
For every process $p \in P$, we define the view function $\text{view}_p$ that maps any sequence $\tau \in (\Delta_{\text{write}} \cup \Delta_{\text{arw}})^*$ of memory changes (due to a write or an atomic read-write transition) to the sequence $\text{view}_p(\tau) \in \left( \bigcup_{x \in X_p} \Delta_{\text{write},x}^p \cup \Delta_{\text{write}}^p \cup \Delta_{\text{arw}}^p \right)^*$ of observable memory changes by the process $p$ (i.e., write and atomic read-write transitions of $p$ and the write operations performed by other processes (without their identity) over $X_p$). Formally, we define $\text{view}_p(\tau)$ in two steps: First we define $\tau_1$ to be $\tau \ominus \left( \bigcup_{p' \neq p} \Delta_{\text{write},x}^{p'} \cup \Delta_{\text{arw},x}^{p'} \right)$ where we remove from $\tau$ all any transition of a process $p'$ different from $p$ accessing a variable that will not be read by $p$. Then, we define $\tau_2 = \tau_1[t/t']t'\in\Delta_{\text{p}}$ to be the sequence constructed from $\tau_1$ by substituting any transition $t$ of a process different from $p$ by its operation $[t]$. Finally, we define from $\text{view}_p(\tau) = \tau_2[\text{arw}(x,v,v')/\text{write}(x,v')]$ where we replace any $\text{arw}$ operation in $\tau_2$ by it corresponding write of the memory.

### 3.5.2 Properties of the Environment Processes

In the following, we give the properties that the environment processes should satisfy.

Let $E = \langle P', A' \rangle$ where $P' = \{p\} \cup \text{EnvSet}$ and $A' = \{ A_p \} \cup \{ A_e \mid e \in \text{EnvSet} \}$.

$E$ is said to be an environment abstraction of $P$ wrt. $p$ if and only if for every singly TSO run $\pi$ wrt. $p$ of $P$ there is a singly TSO run $\pi'$ wrt. $p$ of $E$ such that the following conditions are satisfied:

- $\text{ProgOrder}(\pi)(p) = \text{ProgOrder}(\pi')(p)$.

- $\text{view}_pX_p(\text{TSOrder}(\pi)) = \text{view}_pX_p(\text{TSOrder}(\pi'))$.

We can show that if these properties are satisfied then the persistence of $E$ wrt. $p$ implies the persistence of $P$ wrt. $p$.

**Theorem 5.** Let $E$ be a environment abstraction of $P$. If $P$ has a fragile singly TSO run wrt. $p$ then $E$ has also a fragile singly TSO run wrt. $p$. 39
Proof. We show the theorem by contradiction. Assume that \( P \) has a fragile singly TSO run wrt. \( p \) and \( E \) has no fragile singly TSO run wrt. \( p \). Suppose that \( P \) has a fragile singly TSO run \( \pi \) wrt. \( p \). From the definition of the environment abstract, we know that there is a singly TSO run \( \pi' \) wrt. \( p \) of \( E \) such that:

- \( \text{ProgOrder} (\pi) (p) = \text{ProgOrder} (\pi') (p) \).

- \( \text{view}_p X_p (\text{TSOrder} (\pi)) = \text{view}_p X_p (\text{TSOrder} (\pi')) \).

Since \( E \) has no fragile singly TSO run wrt. \( p \), we know that there is a SC run \( \pi'' \) of \( E \) such that \( \text{trace} (\pi'') = \text{trace} (\pi') \). Now, we can use \( \pi'' \) and \( \pi \) to construct an sequential run \( \pi''' \) of \( p \) such that \( \text{trace} (\pi) = \text{trace} (\pi''') \) and which contradicts the assumption.

Let us assume that \( \pi \) can be rewritten in the form \( \pi = \rho_0 \cdot t_1 \cdot \rho_1 \cdot t_2 \cdot \rho_2 \cdot \cdots \cdot \rho_{n-1} \cdot t_n \cdot \rho_n \)
where \( \text{view}_p X_p (\text{TSOrder} (\rho_j)) = \epsilon \), for \( j : 0 \leq j \leq n \), and \( |\text{view}_p X_p (\text{TSOrder} (t_i))| = 1 \)
for \( i : 1 \leq i \leq n \). In other words, we divide \( \pi \) into segments separated either by \text{arw} \ or \text{write} \ transitions visible to \( p \). Since \( \text{view}_p X_p (\text{TSOrder} (\pi)) = \text{view}_p X_p (\text{TSOrder} (\pi')) \)
and \( \text{trace} (\pi'') = \text{trace} (\pi') \), we can rewrite the run \( \pi'' \) as follows: \( \pi'' = \rho'_0 \cdot t'_1 \cdot \rho'_1 \cdot t'_2 \cdot \rho'_2 \cdot \cdots \cdot \rho'_{n-1} \cdot t'_n \cdot \rho'_n \)
such that \( \text{view}_p X_p (\text{TSOrder} (\rho'_j)) = \epsilon \), for \( j : 0 \leq j \leq n \), and \( |\text{view}_p X_p (\text{TSOrder} (t'_i))| = 1 \).

Let \( \sigma_i = \rho_i \cap \Delta_p \) and \( \sigma'_i = \rho'_i \cap \Delta_p \) for all \( i : 0 \leq i \leq n \). Define \( \pi''' = \rho''_0 \cdot t_1 \cdot \rho''_1 \cdot t_2 \cdot \rho''_2 \cdot \cdots \cdot \rho''_{n-1} \cdot t_n \cdot \rho''_n \) of \( p \) as follows: (1) \( \rho''_n = \sigma_n \cdot \rho'_n \), and (2) For every \( i : 0 < i \leq n \), if \( t_i \in \Delta_p \) then \( \rho''_{i-1} = \sigma_{i-1} \cdot \sigma'_{i-1} \), otherwise \( \rho''_{i-1} = \sigma'_{i-1} \).

Lemma 8. The following properties hold:

- \( \pi''' \) is a sequential run of \( P \).

- \( \text{trace} (\pi) = \text{trace} (\pi''') \).
3.5.3 Environment Instances

In the following, we give two examples of environment abstraction. We use an example with the source code in Figure 3.1 and its automaton in Figure 3.2.

```
// process [1]:
1. x=0;
2. if y==0 goto 1;
// process [2]:
1. y=1;
2. if x==0 goto 1;
```

**Figure 3.1:** A simple example.

**Figure 3.2:** Automata representing the processes of Figure 3.1.

3.5.3.1 Merging all states

**Figure 3.3:** An environment abstraction wrt. \( p \) of the program given in Figure 3.2.

We can define an environment process \( e \) for the set of processes \( P \setminus \{p\} \) by merging all their states into one and adding a self loop transition labeled by any write or atomic read-write operation over \( X_p \) that labels a transition of a process in \( P \setminus \{p\} \). Formally, \( A_e \) is defined by the tuple \( \langle q_e, q_e, \Delta_e \rangle \) where \( q_e \) is the only state of the environment and \( \Delta_e \) is a finite set of transitions such that \( \langle q_e, op, q_e \rangle \) if and only if there is a process \( p' \in P \setminus \{p\} \) and a transition \( t \in \bigcup_{x \in X_p} \Delta_{p'}^{\text{write},x} \cup \Delta_{p'}^{\text{arw},x} \) such that \([t] = \text{op}\).
Lemma 9. \( E = \langle P', A' \rangle \) where \( P' = \{ p, e \} \) and \( A' = \{ A_p, A_e \} \) is an environment abstract wrt. \( p \).

3.5.3.2 Keeping write and atomic read-write transitions

![Diagram](image.png)

Figure 3.4: The environment abstraction wrt. \( p \) of the program given in Figure 3.2 after removing all \( \text{nop} \) transitions, \( k = 1 \).

Let \( k \in \mathbb{N} \) be a natural number. We can define an environment process \( e(p', k) \) for each process \( p' \in P \setminus \{ p \} \) by: (1) by keeping exact all the paths of \( A_{p'} \) up to length \( k \in \mathbb{N} \); and for any read, fence, write, or atomic read-write transitions over \( X_p \) occurring at a position larger than \( k \), we replace it by a \( \text{nop} \) transitions. Formally, \( A_{e(p', k)} \) is defined by the tuple \( \langle Q_{p'} \times \{ 0, 1, \ldots, k \}, (\text{init}_{p'}, 0), \Delta_e(p', k) \rangle \) where \( \Delta_e(p', k) \) is a finite set of transitions such that: (1) For every \( i : 0 \leq i < k, \langle (q, i), \text{op}, (q', i + 1) \rangle \) is in \( \Delta_e(p', k) \) if and only if \( \langle q, \text{op}, q' \rangle \in \Delta_{p'} \), (2) \( \langle (q, k), \text{nop}, (q', k) \rangle \) in \( \Delta_e(p', k) \) if and only if \( \langle q, \text{op}, q' \rangle \in \Delta_{p'}^{\text{read}} \cup \Delta_{p'}^{\text{fence}} \cup \bigcup_{x \in X_p} (\Delta_{p'}^{\text{write}, x} \cup \Delta_{p'}^{\text{aw}, x}) \), and (3) \( \langle (q, k), \text{op}, (q', k) \rangle \) in \( \Delta_e(p', k) \) if and only if \( \langle q, \text{op}, q' \rangle \in \Delta_{p'}^{\text{nop}} \cup \bigcup_{x \in X_p} (\Delta_{p'}^{\text{write}, x} \cup \Delta_{p'}^{\text{aw}, x}) \).

Furthermore, we can apply to the standard automaton techniques to remove all the \( \text{nop} \) transitions in \( A_{e(p', k)} \).

Lemma 10. \( E = \langle P', A' \rangle \) where \( P' = \{ p \} \cup \{ e(p', k) | p' \in P \setminus \{ p \} \} \) and \( A' = \{ A_p \} \cup \{ A_{e(p', k)} | p' \in P \setminus \{ p \} \} \) is an environment abstract wrt. \( p \).
Experiment

We tested 4 types of concurrent algorithms (mutex algorithms and non-blocking write protocols, work stealing queues, idempotent work stealing queues, and concurrent data structures) in [8, 7, 23, 19, 18, 9] with our tool Persist on a laptop machine with core i7 2.4 Ghz and 4 GB memory. Finally, we test thread abstraction feature by increasing the number of processes in Sequential Lock [5], Burns [24], Dijkstra [24], Double-checked locking [31], and an extended version of Dekker mutex exclusion algorithm.

4.1 Mutex Algorithms and Non-blocking Write Protocols

We compared our tool Persist with Trencher [6] for 15 mutex algorithms and non-blocking write protocols (see Table 4.1). Most of them are well-known algorithms such as Dijkstra [24], Dekker [12], Lamport [20, 22], Peterson [30], CLH lock [25], etc.

- For non-persistent/non-robust mutex algorithms under TSO semantics, Persist and Trencher detect them. This means that we can find mutex algo-
algorithms that is potentially dangerous under TSO semantics.

- Then **Persist** and **Trencher** can find a minimal fence set to guarantee persistence/robustness for a non-persistent/non-robust algorithms. Interestingly, **Persist** and **Trencher** generate the same size for minimal fence sets in most of tested algorithms. However, because our violated pattern is simpler than that of **Trencher**, our tool runs faster, especially in big algorithms such as Dekker [12], Lamport Fast [22], or Lamport Bakery [20].

- **MCS Lock** [15], **Nbw-w-ar-ra** [17], and **Nbw-w-br-rl** [17] programs are robust according to **Trencher**, but non-persistent according to our persistence definition. This means that in some cases, our tools generates more fences than **Trencher** does.

- **Nbw-w-wr** [17] programs are persistent by our tool, but non-robust by **Trencher**. This means that in some cases, our tool generates fewer fences than **Trencher** does.

- In most cases, persistence/robustness checking is faster than fence insertion. The reason is that to insert fence, we need to check persistence/robustness. However when we do fence insertion, we know exactly what attacks are, and the instrumented source code will be done only for these attacks. If we do persistence/robustness checking, we instrument the source code to nondeterministically select an attacker, and a pair of `<write, read>` operations. This explains the fact that in some cases, persistence/robustness checking is slower than fence insertion, for example in the **Spin Lock** algorithm [28].
<table>
<thead>
<tr>
<th>Program</th>
<th>T</th>
<th>L</th>
<th>I</th>
<th>Persistence/Robustness checking</th>
<th>Fence insertion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Real</td>
<td>Robust</td>
</tr>
<tr>
<td>Dekker [12]</td>
<td>2</td>
<td>24</td>
<td>30</td>
<td>0.48</td>
<td>No</td>
</tr>
<tr>
<td>Dijkstra [24]</td>
<td>2</td>
<td>22</td>
<td>28</td>
<td>0.47</td>
<td>No</td>
</tr>
<tr>
<td>Burns [24]</td>
<td>2</td>
<td>11</td>
<td>14</td>
<td>0.39</td>
<td>No</td>
</tr>
<tr>
<td>Peterson [30]</td>
<td>2</td>
<td>14</td>
<td>18</td>
<td>0.37</td>
<td>No</td>
</tr>
<tr>
<td>Lamport Bakery [20]</td>
<td>2</td>
<td>20</td>
<td>24</td>
<td>0.53</td>
<td>No</td>
</tr>
<tr>
<td>Lamport Fast [22]</td>
<td>3</td>
<td>33</td>
<td>36</td>
<td>0.54</td>
<td>No</td>
</tr>
<tr>
<td>Parker [11]</td>
<td>2</td>
<td>9</td>
<td>8</td>
<td>0.31</td>
<td>No</td>
</tr>
<tr>
<td>CLH Lock [25]</td>
<td>7</td>
<td>62</td>
<td>58</td>
<td>0.95</td>
<td>No</td>
</tr>
<tr>
<td>JCS Lock [15]</td>
<td>4</td>
<td>30</td>
<td>54</td>
<td>0.70</td>
<td>Yes</td>
</tr>
<tr>
<td>Nbw-w-ar-ra [17]</td>
<td>3</td>
<td>25</td>
<td>22</td>
<td>0.47</td>
<td>Yes</td>
</tr>
<tr>
<td>Nbw-w-br-rl [17]</td>
<td>4</td>
<td>45</td>
<td>45</td>
<td>0.70</td>
<td>Yes</td>
</tr>
<tr>
<td>Nbw-w-r [17]</td>
<td>2</td>
<td>15</td>
<td>13</td>
<td>0.38</td>
<td>Yes</td>
</tr>
<tr>
<td>Spin Lock [28]</td>
<td>1</td>
<td>21</td>
<td>19</td>
<td>0.46</td>
<td>No</td>
</tr>
<tr>
<td>Ticket Spin Lock [28]</td>
<td>2</td>
<td>18</td>
<td>22</td>
<td>0.44</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.1: Mutex algorithms and Non-blocking write protocols.
<table>
<thead>
<tr>
<th>Column</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Name of testing algorithm.</td>
</tr>
<tr>
<td>T</td>
<td>Number of processes.</td>
</tr>
<tr>
<td>L</td>
<td>Number of labels (states).</td>
</tr>
<tr>
<td>I</td>
<td>Number of instructions.</td>
</tr>
<tr>
<td>Persistent/Robust</td>
<td>The program is persistent/robust?</td>
</tr>
<tr>
<td>RQ</td>
<td>Number of reachability queries (RQ = NR1 + NR2 + R).</td>
</tr>
<tr>
<td>NR1</td>
<td>Number of negative reachability queries.</td>
</tr>
<tr>
<td>NR2</td>
<td>Number of negative reachability queries detected by Spin.</td>
</tr>
<tr>
<td>R</td>
<td>Number of positive reachability queries detected by Spin.</td>
</tr>
<tr>
<td>F</td>
<td>Number of fences.</td>
</tr>
<tr>
<td>Spin</td>
<td>Running time in seconds by Spin (translate from Promela file to C file).</td>
</tr>
<tr>
<td>Ver</td>
<td>Verification time in seconds.</td>
</tr>
<tr>
<td>Real</td>
<td>Total time in seconds in parallel.</td>
</tr>
</tbody>
</table>

Table 4.2: Column information.

4.2 Work Stealing Queues

Work stealing queues are important concurrent data structures to optimize the performance of parallel systems by load balancing. We compared our tool Persist with Trencher and DFence [23] for 5 work stealing queue algorithms: Chase-Lev WSQ [10], Cilk’s THE WQS [13], FIFO WSQ [23], LIFO WSQ [23], and Anchor WSQ [23] (see Table 4.3). DFence provides a monitoring approach to find non sequential consistent executions and correct them by inserting fences.

- Persist, Trencher, and DFence detect the same non-persistent/non-robust work stealing queue algorithms.

- Persist, Trencher, and DFence insert fences with the same size.

4.3 Idempotent Work Stealing Queues

In normal work stealing queues, a task must be executed exactly once by a processor. But in idempotent work stealing queues, a task must be executed by at least once. We compared our tool Persist with Trencher and DFence for 3 work stealing
queue algorithms: FIFO iWSQ [27], LIFO iWSQ [27], and Anchor iWSQ [27] (see Table 4.4).

- For idempotent work stealing queues DFENCE does not have specifications for these algorithms, then DFENCE cannot verify them. “-” means that DFENCE does not provide any results.

- PERSIST and TRENCHER think that theses algorithms are non-persistent/non-robust, according to PERSIST and TRENCHER persistence/robustness definitions. However according to [27], these algorithms are safe under TSO semantics. We can explain this because PERSIST and TRENCHER definitions are stronger than sequential consistent requirement.
4.4 Concurrent Data Structures

We have two concurrent data structures, stack and queue, in which there are some processes trying to pop and other processes trying to push elements. We compared our tool Persist with Trencher and DFence for 5 concurrent data structures: MSN queue [26], MS2 queue [26], Lazy list [14], Treiber [35], and Lock free stack [15] (see Table 4.5).

4.5 Thread Abstraction

When we have many processes or some processes have large source code, precise verification is slow. In these cases, thread abstraction can provide us a fence set to guarantee the correctness of programs with a better performance. Let see the Sequential Lock algorithm [5] in Figure 4.3. This algorithm ensures the consistence.
between \textit{data1} and \textit{data2} in readers, and provide a reader-writer mechanism that can avoid the problem of writer starvation. Because there are some data races, the algorithm can be incorrect in TSO models. With different number of writers and readers, we want to check whether the algorithm correct or incorrect and insert fences if needed.

\begin{verbatim}
atomic<unsigned> seq;
int data1, data2;

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    do {
        seq0 = seq;
        r1 = data1;
        r2 = data2;
        seq1 = seq;
    } while (seq0 != seq1 || seq0 & 1);
    //do something with r1 and r2;
}

void writer(...) {
    unsigned seq0 = seq;
    while (seq0 & 1 || !seq.compare_exchange_weak(seq0, seq0+1)) {} 
    data1 = ...;
    data2 = ...;
    seq = seq0 + 2;
}
\end{verbatim}

\textbf{Figure 4.3}: Sequential Lock in [5].

From Table 4.6, we see that TRENCHER, PERSIST with/without thread abstraction can find that this algorithm is correct under TSO semantics.

- When we increase the number of processes, the verification progress needs more time. The reason is that PERSIST and TRENCHER need to verify the state space by interleaving executions of processes. So when we increase the number of readers or writers, the number of interleaving increases, so the state space increases.

- Interestingly, the performance of TRENCHER and PERSIST without thread ab-
straction decrease dramatically fast, but the performance of Persist with thread abstraction is quite stable. This is because of the fact that thread abstraction do abstract for helpers. In this example, reader does not write to memory. So when we do abstraction, the source code of reader contains only \texttt{nop} operations. The increasing number of readers does not effect to the verification process.

- If we increase the number of writers, one of writer will be attacker and other writers will be helpers. This will increase running time in all cases. But when the number of readers is large, the thread abstraction will still be the best.

From this example and more tests in Table 4.8, Table 4.9, Table 4.10, and Figure 4.4, we see that thread abstraction is helpful for us in these cases when we need to guarantee the correctness of programs with a reasonable performance. The extended version of Dekker guarantees partial mutex exclusion by which in a ring topology of many processes, a neighbor of 3 processes has at most one process in critical sections. So a process needs to check whether at least one of its two neighbor processes is in critical section or not. If not, it can enter its critical section.
<table>
<thead>
<tr>
<th>Program</th>
<th>T</th>
<th>L</th>
<th>I</th>
<th>DFence</th>
<th>Trencher</th>
<th>Persist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chase-Lev WSQ [10]</td>
<td>2</td>
<td>39</td>
<td>48</td>
<td>no 1</td>
<td>1.36</td>
<td>1.14</td>
</tr>
<tr>
<td>Cilk's THE WQS [14]</td>
<td>2</td>
<td>66</td>
<td>73</td>
<td>no 2</td>
<td>2.13</td>
<td>0.73</td>
</tr>
<tr>
<td>FIFO WSQ [23]</td>
<td>2</td>
<td>32</td>
<td>12</td>
<td>yes 0</td>
<td>0.14</td>
<td>0.10</td>
</tr>
<tr>
<td>LIFO WSQ [23]</td>
<td>2</td>
<td>37</td>
<td>46</td>
<td>yes 0</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Anchor WSQ [23]</td>
<td>2</td>
<td>54</td>
<td>69</td>
<td>yes 0</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 4.3: Work stealing queues.

<table>
<thead>
<tr>
<th>Program</th>
<th>T</th>
<th>L</th>
<th>I</th>
<th>DFence</th>
<th>Trencher</th>
<th>Persist</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO iWSQ [27]</td>
<td>2</td>
<td>23</td>
<td>29</td>
<td>-</td>
<td>1.90</td>
<td>0.95</td>
</tr>
<tr>
<td>LIFO iWSQ [27]</td>
<td>2</td>
<td>27</td>
<td>37</td>
<td>-</td>
<td>3.60</td>
<td>1.61</td>
</tr>
<tr>
<td>Anchor iWSQ [27]</td>
<td>2</td>
<td>39</td>
<td>48</td>
<td>-</td>
<td>4.45</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Table 4.4: Idempotent work stealing queues.

<table>
<thead>
<tr>
<th>Program</th>
<th>T</th>
<th>L</th>
<th>I</th>
<th>DFence</th>
<th>Trencher</th>
<th>Persist</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSN queue [26]</td>
<td>2</td>
<td>33</td>
<td>63</td>
<td>yes 0</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>MS2 queue [26]</td>
<td>2</td>
<td>31</td>
<td>32</td>
<td>yes 0</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Lazy list [14]</td>
<td>2</td>
<td>81</td>
<td>87</td>
<td>yes 0</td>
<td>0.04</td>
<td>0.03</td>
</tr>
<tr>
<td>Treiber [35]</td>
<td>2</td>
<td>14</td>
<td>16</td>
<td>yes 0</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Lock free stack [15]</td>
<td>4</td>
<td>46</td>
<td>50</td>
<td>unknown</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 4.5: Concurrent data structures. "unknown" means that this test is not tested in paper.
<table>
<thead>
<tr>
<th>No. Writers</th>
<th>No. Readers</th>
<th>TRENCHER with Abstraction</th>
<th>TRENCHER without Abstraction</th>
<th>PERSIST with Abstraction</th>
<th>PERSIST without Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td>F</td>
<td>Real</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.41</td>
<td>0</td>
<td>0.38</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0.58</td>
<td>0</td>
<td>0.42</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>355.19</td>
<td>0</td>
<td>4.05</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1518.59</td>
<td>0</td>
<td>772.05</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>2066.88</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>120</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>150</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>200</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2.21</td>
<td>0</td>
<td>0.80</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3.24</td>
<td>0</td>
<td>0.97</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1510.03</td>
<td>0</td>
<td>412.89</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>1772.44</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
</tr>
<tr>
<td>2</td>
<td>120</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
</tr>
<tr>
<td>2</td>
<td>150</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Table 4.6: Verification Sequential Lock in [5].
<table>
<thead>
<tr>
<th>T/L/I</th>
<th>Trencher</th>
<th>Persist without Thread Abstraction</th>
<th>Persist with Thread Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Real</td>
<td>F</td>
<td>Real</td>
</tr>
<tr>
<td>2/32/40</td>
<td>4.92</td>
<td>4</td>
<td>3.58</td>
</tr>
<tr>
<td>3/48/60</td>
<td>14.12</td>
<td>6</td>
<td>7.32</td>
</tr>
<tr>
<td>4/64/80</td>
<td>973.00</td>
<td>8</td>
<td>121.38</td>
</tr>
<tr>
<td>5/80/100</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>1025.28</td>
</tr>
<tr>
<td>6/96/120</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>2120.12</td>
</tr>
<tr>
<td>7/112/140</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
</tr>
</tbody>
</table>

Table 4.7: Verification Dijkstra [24].

<table>
<thead>
<tr>
<th>T/L/I</th>
<th>Trencher</th>
<th>Persist without Thread Abstraction</th>
<th>Persist with Thread Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Real</td>
<td>F</td>
<td>Real</td>
</tr>
<tr>
<td>2/38/48</td>
<td>5.69</td>
<td>3</td>
<td>4.44</td>
</tr>
<tr>
<td>3/37/72</td>
<td>10.41</td>
<td>5</td>
<td>4.42</td>
</tr>
<tr>
<td>4/76/96</td>
<td>25.83</td>
<td>7</td>
<td>12.60</td>
</tr>
<tr>
<td>5/75/129</td>
<td>896.47</td>
<td>9</td>
<td>131.45</td>
</tr>
<tr>
<td>6/114/144</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>873.79</td>
</tr>
<tr>
<td>7/133/168</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>989.78</td>
</tr>
<tr>
<td>8/152/192</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
</tr>
</tbody>
</table>

Table 4.8: Verification Burns [24].
<table>
<thead>
<tr>
<th>T/L/I</th>
<th>Trencher</th>
<th>Persist without Thread Abstraction</th>
<th>Persist with Thread Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Real</td>
<td>F</td>
<td>Real</td>
</tr>
<tr>
<td>2/26/28</td>
<td>1.41</td>
<td>0</td>
<td>1.33</td>
</tr>
<tr>
<td>3/39/42</td>
<td>2.73</td>
<td>0</td>
<td>1.82</td>
</tr>
<tr>
<td>4/52/56</td>
<td>4.85</td>
<td>0</td>
<td>2.76</td>
</tr>
<tr>
<td>5/65/70</td>
<td>11.76</td>
<td>0</td>
<td>4.26</td>
</tr>
<tr>
<td>6/78/84</td>
<td>64.34</td>
<td>0</td>
<td>8.56</td>
</tr>
<tr>
<td>7/91/98</td>
<td>540.19</td>
<td>0</td>
<td>28.36</td>
</tr>
<tr>
<td>8/104/112</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>115.70</td>
</tr>
<tr>
<td>9/117/126</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>551.87</td>
</tr>
<tr>
<td>10/130/140</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>2065.60</td>
</tr>
</tbody>
</table>

Table 4.9: Verification Double-checked locking [31].

<table>
<thead>
<tr>
<th>T/L/I</th>
<th>Trencher</th>
<th>Persist without Thread Abstraction</th>
<th>Persist with Thread Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Real</td>
<td>F</td>
<td>Real</td>
</tr>
<tr>
<td>3/39/45</td>
<td>15.07</td>
<td>6</td>
<td>11.88</td>
</tr>
<tr>
<td>4/52/60</td>
<td>68.35</td>
<td>8</td>
<td>19.62</td>
</tr>
<tr>
<td>5/65/75</td>
<td>1812.12</td>
<td>10</td>
<td>51.33</td>
</tr>
<tr>
<td>6/78/90</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>485.59</td>
</tr>
<tr>
<td>7/91/105</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>1766.70</td>
</tr>
<tr>
<td>8/104/120</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
</tr>
<tr>
<td>9/117/135</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
</tr>
<tr>
<td>10/130/150</td>
<td>&gt; 40 minutes</td>
<td>unknown</td>
<td>&gt; 40 minutes</td>
</tr>
</tbody>
</table>

Table 4.10: Verification an extended version of Dekker.
Figure 4.4: Persist without thread abstraction outperforms Trencher, and Persist with thread abstraction outperforms both of them while still inferring the minimal fence set.
Conclusions and Future Work

We have presented for the first time, a tool for automatic fence placement that is able to break the scalability barrier both concerning the added complexity due to the presence of event reordering, and also concerning the number of threads that participate in the execution of the program. We have implemented our framework in a prototype and applied it to several challenging examples. In particular, we have verified several examples that cannot be handled by existing approaches. Our work opens several directions:

- Studying new stability definitions by weakening the set of constraints imposed by existing definitions. We are currently considering a stability definition based on removing the source relation from the robustness definition [33] which would lead to a weaker stability definition than robustness and persistence.

- Develop an uniform framework for the persistence detection problem for other weak memory models as done in [3].

- Extend Persist to handle infinite-state programs by using model-checkers for infinite-state systems as back-end tools.
Bibliography


