IT 13 086
Examensarbete 15 hp
November 2013

Sound and Complete Reachability Analysis under PSO

Magnus Lång

Abstract

Sound and Complete Reachability Analysis under PSO

Magnus Lång

Modern multiprocessor systems use weak (relaxed) memory models in order to execute memory sharing multi-threaded code in an efficient manner, but are much harder for programmers to reason about than systems using the sequential consistency memory model.
The SB abstraction and its implementation in the Memorax tool allows sound and complete checking of control state reachability under the TSO memory model, used in modern x86 processors.
In this paper, I present a formalisation of the PSO memory model using the semantics of the Sun SPARC documentation and an alternate semantic, called Partial Write Serialisation, I conjecture to be equivalent with my formalisation under the control state reachability problem. PWS is proved to be a well-structured system which allows sound and complete reachability analysis. An implementation of PWS is presented as part of the Memorax tool and demonstrated experimentally to be capable of analysing reachability and inferring minimal fence sets on many non-trivial and real world examples in reasonable time and memory usage.

Handledare: Mohamed Faouzi Atig
Ämnesgranskare: Parosh Abdulla
Examinator: Olle Gällmo
IT 13 086
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

Weak (*relaxed*) *memory models* [4] allow multiprocessor systems to not provide full *sequential consistency* (SC) — that the final result is equal to that of some interleaving of instructions, executed one by one — to multiple threads accessing the same memory. This is used in most multiprocessor architectures because a requirement on sequential consistency would hamper the ability of the processor to use pipelining and out-of-order execution with memory-accessing code to hide memory latency [10]. *Total Store Order* (TSO) is a common memory model, used by f.ex. Intel x86 [17] among others, and allows the visibility to other threads of a non-atomic write instruction to be delayed past read and other non-memory instructions. The *Partial Store Order* (PSO) memory model, used in Sun SPARC [18], allows non-atomic write instructions to be delayed past any other instruction except for other writes to the same address.

In recent years, lock-free, or non-blocking, algorithms, i.e. algorithms that synchronise access from several threads without making use of hardware or operating system mutual exclusion primitives, have become popular because of their performance and flexibility. However, weak memory models make reasoning about the correctness of these algorithms hard, and many algorithms that are correct under sequential consistency requires the use of special *fence* instructions, also known as *memory barriers*, that prevent the behaviours incompatible with SC [4]. Fence instructions decrease the performance of the algorithms [9], so the minimal amount necessary to make an algorithm correct under some memory model is often desired.

The SB abstraction [2], and its implementation in the Memorax [3] tool provides sound and complete control state reachability analysis and inference of all minimal sets of fence instructions required to make an algorithm fulfil a safety property under TSO. In this paper, I will present a formalisation of the PSO memory model based on previous work and on the documentation of the SPARC architecture, an abstraction akin to SB, called *Partial Write Serialisation*, that allows for sound and complete control state reachability analysis under PSO as well as an implementation of this abstraction as part of the Memorax tool.

1.1 Related works

To my knowledge, my approach is the first sound and complete control state reachability analysis method for programs running under PSO. Most previous work have focused on different memory models than PSO and is thus not directly comparable. One noteworthy work is that of Alexander Linden and Pierre Wolper [16] that presents a sound but incomplete analysis method for PSO. Other approaches have made use of partial-coherence abstractions [15] which results in over-approximations of PSO that might incorrectly claim that some state is reachable and compute non-minimal fence sets. Under-approximations of PSO can be achieved through testing [7], using bounded buffers [11, 14], et.c., and can be useful to find errors, but are not able to prove the safety of a program in general. Techniques to insert fences that remove all sequentially inconsistent behaviours even if these will not violate the desired correctness property are monitors [5, 8] and explicit state model checking [13]. These methods cannot guarantee to generate minimal sets of fences to make programs correct, because they also remove benign sequentially inconsistent behaviours.

2 Preliminaries

In this section I describe the notation used in the rest of this paper.

2.1 Notation

For sets A and B, I use $[A \mapsto B]$ to denote the set of all total functions from A to B and $f : A \mapsto B$ to denote that f is a total function that maps A to B. For $a \in A$ and $b \in B$, $f[a \hookleftarrow b]$ denotes the function f' defined as follows:

$$f'(a') = \begin{cases} b & a' = a \\ f(a') & a' \neq a \end{cases}$$

Words Let Σ be a finite alphabet. Σ^* denotes the set of all *words* (resp. Σ^+ the set of all non-empty words) over Σ. Let ε be the empty word. The length of a word $w \in \Sigma^*$ is denoted by $|w|$ and the length of the empty word is $|\varepsilon| = 0$. For every $i : 1 \leq i \leq |w|$, let $w(i)$ be the symbol at position i in w.

For $a \in \Sigma$, I write $a \in w$ if a appears in w, i.e., $a = w(i)$ for some $i : 1 \leq i \leq |w|$. For words w_1, w_2, the concatenation of w_1 and w_2 is written as $w_1 \cdot w_2$. The notation $(\ldots)$ will denote an arbitrary word in Σ^*, and if used multiple times, it refers to *different*, i.e. not necessarily equal, arbitrary words in Σ^*.

Transition Systems

Definition 1. A transition system $\mathcal{T}$ is a triple $(\texttt{C}, \texttt{Init}, \rightarrow)$ where $\texttt{C}$ is a (potentially infinite) set of *configurations*, $\texttt{Init} \subseteq \texttt{C}$ is the set of *initial configurations*, and $\rightarrow \subseteq \texttt{C} \times \Lambda \times \texttt{C}$ is the *transition relation.* Λ is a finite set of labels.

Writing $c \xrightarrow{\lambda} c'$ denotes that $(c, \lambda, c') \in \rightarrow$, writing $c \rightarrow c'$ denotes that $(c, \lambda, c') \in \rightarrow$ for some arbitrary $\lambda \in \Lambda$, and writing $\xrightarrow{*}$ denotes the reflexive transitive closure of $\rightarrow$.

Definition 2. A configuration c is said to be *reachable* if $c_0 \xrightarrow{*} c$ for some $c_0 \in \texttt{Init}$; and a set C of configurations is said to be *reachable* if some $c \in C$ is reachable.

Definition 3. A *run* π of $\mathcal{T}$ is a sequence of configurations of the form $c_0 \rightarrow c_1 \rightarrow \cdots \rightarrow c_n$, where $c_i \rightarrow c_{i+1}$ for all $i : 0 \leq i < n$. It is also written as $c_0 \xrightarrow{\pi} c_n$.

Definition 4. Given an preorder $\sqsubseteq$ on $\texttt{C}$ I say that $\rightarrow$ is *monotonic* wrt. $\sqsubseteq$ if whenever $c_1 \rightarrow c_1'$ and $c_1 \sqsubseteq c_2$, there exists a c_2' s.t. $c_2 \xrightarrow{*} c_2'$ and $c_1' \sqsubseteq c_2'$. The transition relation $\rightarrow$ is *effectively monotonic* wrt. $\sqsubseteq$ if, given configurations c_1, c_1', c_2 as above, c_2' and a run π s.t. $c_2 \xrightarrow{\pi} c_2'$ can be computed.

Definition 5. Define $c\uparrow$ to be the upwards closure of c w.r.t. $\sqsubseteq$ and $S\uparrow$ for $S \subseteq C$ as $\bigcup_{c \in S} c\uparrow$.

Definition 6. The minor set $\texttt{min}(S)$ is the smallest set S' such that $S\uparrow = S'\uparrow$. It follows from the fact that $\sqsubseteq$ is a preorder that $S' \subseteq S$.

Definition 7. For $S \subseteq C$ and $\lambda \in \Lambda$, the predecessor set $\texttt{pre}_\lambda(S)$ is defined to be $\left\{c' \mid c \in S,\, c \xrightarrow{\lambda} c'\right\}$ and $\texttt{pre}(S)$ as $\bigcup_{\lambda \in \Lambda} \texttt{pre}_\lambda(S)$.

2.2 Concurrent Programs

I define *concurrent programs*, a model for representing shared-memory concurrent processes. A concurrent program $\mathcal{P}$ has a finite number of finite-state processes (threads), each with its own program code. Communication between processes is performed through a shared memory that consists of a fixed number of shared variables of finite domains to which all threads can read and write.

I assume a finite set X of *variables* ranging over a finite data domain V.

Definition 8. A *concurrent program* is a pair $\mathcal{P} = (P, A)$ where P is a finite set of *processes* and $A = \{A_p \,|\, p \in P\}$ is a set of extended finite-state automata (one automaton A_p for each process $p \in P$).

The automaton A_p is a triple $\left(Q_p, q_p^{init}, \Delta_p\right)$ where Q_p is a finite set of *local states*, $q_p^{init} \in Q_p$ is the *initial* local state, and Δ_p is a finite set of *transitions*. Each transition is a triple (q, op, q') where $q, q' \in Q_p$ and op is an *operation*.

Definition 9. An operation is of one of the following six forms:

(1) *"no operation"* `nop`

(2) *read operation* $\mathtt{r}\,(x, v)$

(3) *write operation* $\mathtt{w}\,(x, v)$

(4) *store-load fence operation* `mfence`

(5) *store-store fence operation* `sfence`

(6) *atomic read-write operation* $\mathtt{arw}\,(x, v, v')$

where $x \in X$ and $v, v' \in V$.

Definition 10. Define $Q \coloneqq \bigcup_{p \in P} Q_p$ and $\Delta \coloneqq \bigcup_{p \in P} \Delta_p$. A *local state definition* $\underline{q}$ is a mapping $P \mapsto Q$ such that $\underline{q}\,(p) \in Q_p$ for each $p \in P$.

3 The Problem

In order to design a method for sound and complete reachability analysis under PSO, I will be using a previous work[2] that presented a method for sound and complete reachability analysis under TSO as a starting point.

This method makes use of the theory of *well-structured systems*[1]

Definition 11. A well-founded preorder $\sqsubseteq$ on X is called a *well-quasi ordering* if there is no infinite sequence $x_0, x_1, \ldots \in X$ such that $x_i \not\sqsubseteq x_j$ for all $i < j$.

Definition 12. A transition system $\mathcal{T} = (\mathtt{C}, \mathtt{Init}, \rightarrow)$, assuming a decidable well-founded relation $\sqsubseteq$ on $\mathtt{C}$, is said to be *well-structured* if

(1) it is monotonic w.r.t. $\sqsubseteq$;

(2) $\sqsubseteq$ is a *well-quasi ordering*; and

(3) for each configuration $c \in \mathtt{C}$ and $\lambda \in \Lambda$, the set $\mathtt{min}\,(\mathtt{pre}_\lambda\,(c\uparrow))$ is computable.

[2] does not define a well-quasi-ordering over the set of TSO configurations, however. Rather, they present an alternate semantic called *Single-Buffer* (SB), that has the property that for any concurrent program, the set of reachable local state definitions is the same for TSO as it is for SB. Then they define a well-quasi-order over SB that makes it a well-structured system, which allows for sound and complete reachability analysis [1, p. 6].

To design an equivalent method for PSO, a formalisation of the memory model is required. It cannot be arbitrarily chosen because if it does not describe the behaviour of PSO, the analysis will be useless.

I will then attempt to design a monotonic well-quasi-order over that model, and presumably fail (indeed, if I would succeed, then that well-quasi order could be used to analyse TSO, assuming that TSO can be simulated with PSO by inserting `sfence` instructions between normal transitions of a program, rendering SB obsolete). The nature of this failure will guide me in designing an alternate semantic for PSO along with a monotonic well-quasi ordering of its configurations.

4 PSO Semantics

I will consider a formalisation of the PSO memory model described by [6] with the store-store fence semantics of [16]. Conceptually, each process is associated with a FIFO buffer for each variable (memory location). This buffer is used to store write operations of the process that have not yet reached main memory and become readable by other processes.

4.1 Formal Semantics

A *PSO-configuration* c is a triple $\left(\underline{q},\, \underline{b},\, mem\right)$, where $\underline{q}$ is a local state definition, $\underline{b} : P \mapsto \left[X \mapsto (V \cup \{\star\})^*\right]$ and $mem : X \mapsto V$. $\underline{q}\,(p)$ gives the local state of process p, $mem\,(x)$ is the value of variable x and $\underline{b}\,(p)\,(x)$ is the FIFO-buffer of writes to variable x by process p that have yet to reach memory. The buffer may also contain the *store-store fence* symbol $\star$ that restrict the reordering of stores.

The set of PSO-configurations is denoted by $\mathtt{C}_{PSO}$. The transition relation $\rightarrow_{PSO}$ on $\mathtt{C}_{PSO}$ is defined as the union of

(1) the members of Δ; and

(2) a set $\Delta' := \left\{\mathtt{update}_{p,\,x} \,|\, p \in P,\, x \in X\right\} \cup \left\{\mathtt{update}_{p,\,\star} \,|\, p \in P\right\}$ where $\mathtt{update}_{p,\,x}$ is an operation that updates the memory using the first value in the buffer of process p to variable x and $\mathtt{update}_{p,\,\star}$ removes the store-store fence symbol from the head of all the buffers of process p.

For configurations $c = \left(\underline{q},\, \underline{b},\, mem\right)$, $c' = \left(\underline{q}',\, \underline{b}',\, mem'\right)$, a process $p \in P$, local states of process p; $q,\, q' \in Q_p$, a variable $x \in X$, and $t \in \Delta_p \cup \{\mathtt{update}_{p,\,x},\, \mathtt{update}_{p,\,\star}\}$ such that $\underline{q}\,(p) = q$, $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, I write $c \xrightarrow{t}_{PSO} c'$ to denote that one of the following conditions is satisfied:

– Nop: $t = (q,\, \mathtt{nop},\, q')$, $\underline{b}' = \underline{b}$, and $mem' = mem$. Process p transitions to a new state q' whilst the contents of buffers and memory remain unchanged.

– Write to store: $t = (q, \mathtt{w}(x, v), q')$, $\underline{b}' = \underline{b}[p \hookleftarrow \underline{b}(p)[x \hookleftarrow b(p)(x) \cdot v]]$, and $mem' = mem$. The written value v is added to the tail of the buffer for variable x of process p.

– Update: $t = \mathtt{update}_{p,x}$, $q = q'$, $\underline{b} = \underline{b}'\left[p \hookleftarrow \underline{b}'(p)\left[x \hookleftarrow v \cdot \underline{b}'(p)(x)\right]\right]$, $v \in V$, and $mem' = mem[x \hookleftarrow v]$. The oldest value in the write buffer for x of p is removed, and memory is updated accordingly. Note that $v \neq \star$.

– Remove fence symbol: $t = \mathtt{update}_{p,\star}$, $q = q'$,
$\forall x \in X. \underline{b} = \underline{b}'\left[p \hookleftarrow \underline{b}'(p)\left[x \hookleftarrow \star \cdot \underline{b}'(p)(x)\right]\right]$, and $mem' = mem$. The store-store fence symbol $\star$ is removed from the head of all buffers of process p.

– Read: $t = (q, \mathtt{r}(x, v), q')$, $\underline{b}' = \underline{b}$, $mem' = mem$, and one of the following two conditions is satisfied:

  - Read own write: $\underline{b}(p)(x) = (\ldots) \cdot v \cdot F$, $F \in \star^*$. There are writes in the buffer of x for the process p. The most recent such write is considered.
  - Read memory: $\underline{b}(p)(x) \in \star^*$ and $mem(x) = v$. There are no pending writes of x for process p. v is fetched from memory.

– Store-store fence: $t = (q, \mathtt{sfence}, q')$,
$\forall x \in X. \underline{b}' = \underline{b}[p \hookleftarrow \underline{b}(p)[x \hookleftarrow \underline{b}(p)(x) \cdot \star]]$, and $mem' = mem$. A store-store fence symbol $\star$ is added to the tail of all buffers of process p.

– Store-load fence: $t = (q, \mathtt{mfence}, q')$, $\underline{b}' = \underline{b}$, $\forall x \in X. \underline{b}(p)(x) = \varepsilon$, and $mem' = mem$. A store-load fence requires all buffers of the process to be empty.

– ARW: $t = (q, \mathtt{arw}(x, v, v'), q')$, $\underline{b}' = \underline{b}$, $\underline{b}(p)(x) = \varepsilon$, $mem(x) = v$, and $mem' = mem[x \hookleftarrow v']$. The ARW operation corresponds to an atomic compare and swap (or test and set). Because the ARW operation blocks until the write is visible to other processors [18, 8.4.6.1], the write is done to memory directly. And since atomic load-stores is allowed to be overrun by other stores in PSO [18, 8.4.4], I will not require all write buffers of p to be empty, but since SPARC requires processor self-consistency [18, 8.4.1] p:s buffer for x is required to be empty. The operation checks whether the value of x is v. In such a case, it changes its value to v'.

I use $c \rightarrow_{PSO} c'$ to denote that $c \xrightarrow{t}_{PSO} c'$ for some $t \in \Delta \cup \Delta'$. The set $\mathtt{Init}_{PSO}$ of *initial* PSO-configurations contains all configurations of the form $\left(\underline{q_{init}}, \underline{b_{init}}, mem_{init}\right)$ where, for all $p \in P$ and $x \in X$, we have $\underline{q_{init}}(p) = q_p^{init}$ and $\underline{b_{init}}(p)(x) = \varepsilon$. In other words, each process is in its initial local state and all the buffers are empty. On the other hand, the memory may have any initial value. The transition system induced by a concurrent system under the PSO semantics is then given by $(\mathtt{C}_{PSO}, \mathtt{Init}_{PSO}, \rightarrow_{PSO})$.

4.2 The PSO Reachability Problem

Given a set $\mathtt{Target}$ of local state definitions, $Reachable(PSO)(\mathcal{P})(\mathtt{Target})$ is a predicate that indicates the reachability of the set $\left\{\left(\underline{q}, \underline{b}, mem\right) \mid \underline{q} \in \mathtt{Target}\right\}$, i.e. whether a configuration c, where the local state definition of c belongs to `Target`, is reachable. The reachability problem for PSO is to check, for a given `Target`, whether $\mathit{Reachable}\,(PSO)\,(\mathcal{P})\,(\texttt{Target})$ holds or not. Using standard techniques, checking safety properties can be reduced to the reachability problem. More precisely, `Target` denotes "bad configurations" that are not wanted to occur during the execution of the system. For instance, for mutual exclusion protocols, the bad configurations are those where the local states of two processes are both in the critical sections. I say that the "program is correct" to indicate that `Target` is not reachable.

4.3 Problems of a Well-Quasi Ordering Over $\texttt{C}_{PSO}$

To define a well-quasi ordering (wqo) over $\texttt{C}_{PSO}$, I need to in particular define a wqo over the infinite part of the data domain; the write buffers. SB uses the subsequence relation over its single buffer. Doing so for the write buffers of PSO, however, would not allow the preorder to be monotonic; if the store-store fence symbol $\star$ is inserted in some location in each of the store buffers of a process in a configuration, that configuration can reach fewer states than before. This means that if the two configurations were defined to be ordered, monotony would not hold. The two states cannot be defined to be unrelated either, because that would create an infinite set of states with different counts of stars that are minimal in $\texttt{C}_{PSO}$, violating the requirement of well-quasi orderings that there is no infinite sequence where each element isn't larger than any previous element.

5 Partial Write Serialisation Semantics

I introduce a new semantics model, Partial Write Serialisation (PWS), that avoids the $\star$ symbol which prevented the existence of a wqo on C_{PSO} by allowing serialisation of write order in a single buffer before they are made visible to all processes, in a similar fashion to the SB semantics of [2].

5.1 Formal Semantics

A *PWS-configuration* c is a quadruple $\left(\underline{q},\, \underline{b},\, \mathit{mem},\, \underline{z}\right)$ where $\underline{q}$ is a local state definition, $\underline{b} : P \mapsto [X \mapsto V^*]$ is the per process and variable FIFO buffer, $\mathit{mem} \in ([X \mapsto V] \times P \times X)^+$ is a sequence of memory snapshots, along with the last variable that was written, and which process did so. mem is called the *channel* and its elements *messages*. $\underline{z} : P \mapsto \mathbb{N}$ represents a set of *pointers*, one per process, such that $\mathit{mem}\,(\underline{z}\,(p))$ is the message containing the memory snapshot that the process p is currently observing.

Let $c = \left(\underline{q},\, \underline{b},\, \mathit{mem},\, \underline{z}\right)$ be a PWS-configuration. For every $p \in P$ and $x \in X$, I use $\mathrm{LastWrite}\,(c,\, p,\, x)$ to denote the index of the most recent message where p writes to x or the currently observed message of p, whichever comes last. Formally, $\mathrm{LastWrite}\,(c,\, p,\, x)$ is the largest index i such that $i = \underline{z}\,(p)$ or $\mathit{mem}\,(i) = (m,\, p,\, x)$ for some m.

The transition relation $\rightarrow_{PWS}$ on the set of PWS-configurations is defined as the union of

(1) The members of Δ; and

(2) A set $\Delta'' := \left\{\texttt{serialise}_{p,\,x} \,|\, p \in P,\, x \in X\right\} \cup \left\{\texttt{update}_p \,|\, p \in P\right\}$ where $\texttt{serialise}_{p,\,x}$ is an operation that serialises the oldest write to x by p into a new message at the end of *mem*, and $\texttt{update}_p$ is an operation that advances $\underline{z}\,(p)$ by one, effectively updating memory from the point of p.

Now, for PWS-configurations $c = \left(\underline{q},\, \underline{b},\, mem,\, \underline{z}\right)$ and $c' = \left(\underline{q}',\, \underline{b}',\, mem',\, \underline{z}'\right)$, a process $p \in P$, local states of process p $q,\, q' \in Q_p$, a variable $x \in X$, pointers $z,\, z' \in \mathbb{N}$, and $t \in \Delta_p \cup \left\{\texttt{serialise}_{p,\,x},\, \texttt{update}_p\right\}$ such that $\underline{q}\,(p) = q$, $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, $\underline{z}\,(p) = z$, and $\underline{z}' = \underline{z}\,[p \hookleftarrow z']$, I write $c \xrightarrow{t}_{PSO} c'$ to denote that one of the following conditions is satisfied:

- Nop: $t = (q,\, \texttt{nop},\, q')$, $\underline{b}' = \underline{b}$, $mem' = mem$, and $z' = z$. The operation only changes the local state of p.
- Write to store: $t = (q,\, \texttt{w}\,(x,\, v)\,,\, q')$, $\underline{b}' = \underline{b}\,[p \hookleftarrow \underline{b}\,(p)\,[x \hookleftarrow b\,(p)\,(x) \cdot v]]$, $mem' = mem$, and $z' = z$. The written value is added to the local FIFO buffer of x for process p.
- Serialise: $t = \texttt{serialise}_{p,\,x}$, $q = q'$, $\underline{b} = \underline{b}'\,\left[p \hookleftarrow \underline{b}'\,(p)\,\left[x \hookleftarrow v \cdot \underline{b}'\,(p)\,(x)\right]\right]$, $v \in V$, $mem\,(|mem|) = (m_1,\, p_1,\, x_1)$, $mem' = mem \cdot (m_1\,[x \hookleftarrow v]\,,\, p,\, x)$, and $z' = z$. The oldest write to x in process p's buffer is serialised as a message at the end of *mem*.
- Update: $t = \texttt{update}_p$, $q = q'$, $\underline{b}' = \underline{b}$, $mem' = mem$, $z < |mem|$, $z' = z{+}1$. An update operation (as seen by p) is simulated by moving the pointer of p one step to the right. This makes p observe a more recent message.
- Read: $t = (q,\, \texttt{r}\,(x,\, v)\,,\, q')$, $\underline{b}' = \underline{b}$, $mem' = mem$, $z' = z$, and one of the following two conditions is satisfied:
  - Read own write: $\underline{b}\,(p)\,(x) = (\ldots) \cdot v$.
  - Read memory: For some $m_1,\, p_1,\, x_1$ with $m_1\,(x) = v$; $\underline{b}\,(p)\,(x) = \varepsilon$ and $mem\,(\mathrm{LastWrite}\,(c,\, p,\, x)) = (m_1,\, p_1,\, x_1)$.
- Store-store fence: $t = (q,\, \texttt{sfence},\, q')$, $\underline{b}' = \underline{b}$, $\forall x \in X.\, \underline{b}\,(p)\,(x) = \varepsilon$, $mem' = mem$, and $z' = z$. By requiring all previous writes of p to be serialised before continuing, the writes of p cannot reorder past a `sfence`.
- Store-load fence: $t = (q,\, \texttt{mfence},\, q')$, $\underline{b}' = \underline{b}$, $\forall x \in X.\, \underline{b}\,(p)\,(x) = \varepsilon$, $mem' = mem$, and $z' = z = |mem|$. Every write by p must have been made visible to other processors before continuing, which is encoded by the fact that p's buffers are empty and p is observing the most recent message.
- ARW: $t = (q,\, \texttt{arw}\,(x,\, v,\, v')\,,\, q')$, $\underline{b}' = \underline{b}$, $\underline{b}\,(p)\,(x) = \varepsilon$, $z = |mem|$, $z' = |mem| + 1$, and $mem\,(|mem|) = (m_1,\, p_1,\, x_1)$ for some $m_1,\, p_1,\, x_1$ with $m_1\,(x) = v$, $mem' = mem \cdot (m_1\,[x \hookleftarrow v']\,,\, p,\, x)$. It is required that p is observing the most recent message and that p has no writes to x in it's buffers. To encode the atomicity of the operation, the updated message is immediately added to *mem*.

I use $c \rightarrow_{PWS} c'$ to denote that $c \xrightarrow{t}_{PWS} c'$ for some $t \in \Delta \cup \Delta''$. The set $\texttt{Init}_{PWS}$ of *initial* PWS-configurations contains all configurations of the form $\left(\underline{q_{init}}, \underline{b_{init}}, mem_{init}, \underline{z_{init}}\right)$ where $|mem_{init}| = 1$, and for all $p \in P$ and $x \in X$, we have $\underline{q_{init}}\,(p) = q_p^{init}$, $\underline{b_{init}}\,(p)\,(x) = \varepsilon$, and $\underline{z_{init}}\,(p) = 1$. In other words, each process is in its initial state and all the process and variable local buffers are empty. The channel mem_{init} contains a single message of the form $(m_{init}, p_{init}, x_{init})$, where m_{init} represents the initial value of memory. The memory may have any initial value. Also, the values of p_{init} and x_{init} is not relevant since they will not be used in the computation of the system. The pointers of all the processes point to the first (and only) message. The transition system induced by a concurrent system under PWS semantics is then given by $(\texttt{C}_{PWS}, \texttt{Init}_{PWS}, \rightarrow_{PWS})$.

5.2 The PWS Reachability Problem

The predicate *Reachable* $(PWS)\,(\mathcal{P})\,(\texttt{Target})$, and the reachability problem for the PWS semantics, is defined in a similar manner to PSO. The following conjecture states equivalence of the reachability problems under PSO and PWS semantics. Although I will not prove it here, I claim that the proof is a straight-forward, if tedious to construct, extension of the SB-TSO reachability equivalence theorem in [2].

Conjecture 1. *For a concurrent program $\mathcal{P}$ and a local state definition* `Target`*, the reachability problems are equivalent under the PSO and PWS semantics.*

5.3 Well-Quasi Ordering

For a PWS-configuration $c = \left(\underline{q}, \underline{b}, mem, \underline{z}\right)$, the index of the first message that any process is pointing to is defined as $\text{ActiveIndex}\,(c) := \texttt{min}\,\{\underline{z}\,(p) \mid p \in P\}$.

Given two PWS-configurations $c = \left(\underline{q}, \underline{b}, mem, \underline{z}\right)$ and $c' = \left(\underline{q}', \underline{b}', mem', \underline{z}'\right)$, define $j := \text{ActiveIndex}\,(c)$ and $j' := \text{ActiveIndex}\,(c')$. The relation $c \sqsubseteq c'$ is defined as all the following conditions holding:

(1) $\underline{q} = \underline{q}'$

(2) For every process $p \in P$ and variable $x \in X$, there is an injection $g_{p,\,x} : \{1, \ldots, |\underline{b}\,(p)\,(x)|\} \mapsto \left\{1, \ldots, \left|\underline{b}'\,(p)\,(x)\right|\right\}$ such that for every $i, i_1, i_2 \in \{1, \ldots, |\underline{b}\,(p)\,(x)|\}$

  (a) $i_1 > i_2 \implies g_{p,\,x}\,(i_1) > g_{p,\,x}\,(i_2)$

  (b) $\underline{b}\,(p)\,(x)\,(i) = \underline{b}'\,(p)\,(x)\,(g_{p,\,x}\,(i))$

(3) There is an injection $h : \{j, \ldots, |mem|\} \mapsto \{j', \ldots, |mem'|\}$ such that for every $i, i_1, i_2 \in \{j, \ldots, |mem|\}$ the following conditions hold:

  (a) $i_i > i_2 \implies h\,(i_1) > h\,(i_2)$

  (b) $mem\,(i) = mem'\,(h\,(i))$

  (c) $\underline{z}'\,(p) = h\,(\underline{z}\,(p))$

(4) For every process $p \in P$ and variable $x \in X$, one of the following conditions hold:

(a) $\underline{b}(p)(x) = (\ldots) \cdot v$ and $\underline{b}'(p)(x) = (\ldots) \cdot v$

(b) $\underline{b}(p)(x) = \varepsilon$, $\underline{b}'(p)(x) = \varepsilon$

(5) For every process $p \in P$ and variable $x \in X$, $\mathrm{LastWrite}(c', p, x) = h(\mathrm{LastWrite}(c, p, x))$.

Lemma 1. *The relation $\sqsubseteq$ is a well-quasi ordering on PWS-configurations.*

Proof. This is an immediate consequence of the fact that

(1) The subsequence relation ((2) and (3ab)) is a well-quasi ordering on finite words [12], and that

(2) the number of states (1), pointers (3c), and observed memory state and last writes ((4) and (5)) that should be equal, is finite.

□

Lemma 2. *$\rightarrow_{PWS}$ is effectively monotonic wrt. $\sqsubseteq$.*

Proof. I will now show that given PWS-configurations c_1, c_1', and c_2 such that $c_1 \rightarrow_{PWS} c_1'$ and $c_1 \sqsubseteq c_2$, there exists an PWS-configuration c_2' such that $c_2 \xrightarrow{*}_{PWS} c_2'$ and $c_1' \sqsubseteq c_2'$.

Let h and $g_{p,x}$ be the injections defined by $c_1 \sqsubseteq c_2$. I will consider each operation $t \in \Delta_p \cup \{\mathtt{serialise}_{p,x}, \mathtt{update}_p\}$ for some $p \in P$ and $x \in X$ such that $c_1 \xrightarrow{t}_{PWS} c_1'$ in turn.

- Nop: The $t = (q, \mathtt{nop}, q')$ operation only changes the local state definition $\underline{q}$ of c_1 and c_2 (recall that $c_1 \sqsubseteq c_2$ implies their local state definitions are equal). If the same operation is applied to c_2, it will result in the same local state definition as in c_1', and since no other element of c_1 or c_2 changes, $c_1' \sqsubseteq c_2'$ for $c_2 \xrightarrow{t}_{PWS} c_2'$.

- Write to store: The $t = (q, \mathtt{w}(x, v), q')$ operation, in addition to what nop does, appends a value to p's buffer of x. If the same operation is applied to c_2; $c_2 \xrightarrow{t}_{PWS} c_2'$, the same value is appended to p's buffer of x in c_2, and thus the subsequence condition (2) and the condition that the buffers must end with the same value (4) will still hold. Thus, $c_1' \sqsubseteq c_2'$.

- Read: The $t = (q, \mathtt{r}(x, v), q')$ operation behaves like a nop, but requires process p to observe x having value v. But since that is always true in c_2 if it is in c_1 (because of conditions (4) and (5)), $c_1' \sqsubseteq c_2'$ for $c_2 \xrightarrow{t}_{PWS} c_2'$, as with nop.

- Serialise: The $t = \mathtt{serialise}_{p,x}$ operation takes an element from p's write buffer for x and makes a message of it. The same element exists in c_2 and will make the same message when serialised. However, there might be more elements in c_2 from p to x that must be serialised before that element is reached in c_2. If $\pi = \underbrace{\mathtt{serialise}_{p,x} \rightarrow \cdots \rightarrow \mathtt{serialise}_{p,x}}_{g_{p,x}(1)}$, $c_2 \xrightarrow{\pi}_{PWS} c_2'$ will serialise all p's writes to x in c_2 up to and including the one that corresponds to the one that is being serialised in c_1. Since π removes the buffer element corresponding to the one being removed by t, but not any that corresponds to elements not removed by t, and the same message is created at the end of the channel for both c_1 and c_2, the subsequence conditions of the relation holds. And since neither t nor π change the local state definitions or pointers, the serialise operation changes the LastWrite of only p and x in both configurations to a new, corresponding, message, and the serialised write is either the last write of both configurations, or of neither, $c'_1 \sqsubseteq c'_2$.

– Update: The $t = \texttt{update}_p$ operation advances the pointer of process p to a more recent message. The corresponding message in c_2 might not be immediately following the message currently pointed to by p. However, by performing several updates, the pointer in c_2 can be advanced to point to the message corresponding to the more recent message in c_1. In particular, if z is the old pointer of p in c_1, $\pi = \underbrace{\texttt{update}_p \to \cdots \to \texttt{update}_p}_{h(z+1)-h(z)}$ defines $c_2 \xrightarrow{\pi}_{PWS} c'_2$ where this is the case. Since the pointer of p in c_1 has been advanced from z to $z+1$ and the pointer of p in c_2 from $h(z)$ to $h(z+1)$, (4b) will still hold. Also, none of the additional messages in c_2 is associated with any LastWrites, because otherwise (5) would not hold for $c_1 \sqsubseteq c_2$. Thus $c'_1 \sqsubseteq c'_2$.

– ARW: The $t = (q, \texttt{arw}(x, v, v'), q')$ operation performs both read, write, serialise and update as a single operation, all of which have been shown to be monotonic operations. ARW requires p's write buffer of x to be empty, and thus that it is the case in c_1. Moreover, condition (4) requires this to be the case in c_2 as well. ARW also requires p's pointer to be on the very last message, but it must be the case in c_2 if it is in c_1, because otherwise, the last message of c_2 that doesn't have a corresponding message in c_1 would be the LastWrite for some p and x in c_2, and (5) would not hold. We see that c_2 would not require multiple instructions to serialise or update this write, since I have shown its buffer to be empty and pointer to be at the last message. Thus $c_2 \xrightarrow{t}_{PWS} c'_2$ and also $c'_1 \sqsubseteq c'_2$.

I have now shown that such an c'_2 exists in all cases, and I have also detailed how to compute π such that $c_2 \xrightarrow{\pi}_{PWS} c'_2$ in each of these cases.

□

6 Reachability Analysis

In this section, I will describe how Memorax implements reachability analysis using the framework of [1]. I will describe how PWS-configurations are encoded and how the state explosion problem is addressed.

6.1 Configuration encoding

Memorax encodes PWS-configurations as something called constraints. They are essentially PWS-configurations, but wherever a PWS-configurations has a value from the set V, a PWS-*constraint* has a value from the set $V \cup \{*\}$. The value $*$ signifies an arbitrary value, so, for example, the constraint that has a single message in the channel, where each memory location is mapped to $*$ represents all configurations that are in other aspects equal to the constraint, but has arbitrary values in their only message.

6.2 Basic algorithm

Memorax performs backwards reachability analysis using minor sets of configurations that represent their upwards closure w.r.t. $\sqsubseteq$. Because $\sqsubseteq$ is monotonic, the analysis is exact [1]. Because $\sqsubseteq$ is a well-quasi ordering, the analysis will terminate after a finite number of steps [1, p. 3].

For a constraint c, let $\texttt{premin}_\lambda(c) = \texttt{min}(\texttt{pre}_\lambda(c\uparrow))$. The analysis uses a minor set of constraints F and a queue $Q \subseteq F$. F has an initial value of $\texttt{min}\left(\left\{\left(\underline{q}, \underline{b}, mem, \underline{z}\right) \mid \underline{q} \in forbidden\right\}\right)$, where $forbidden$ is the set of local state definitions whose reachability is being computed, and Q has the same initial value.

Each step, a constraint c is popped from Q and for each $t \in \Delta_p \cup \left\{\texttt{serialise}_{p,x}, \texttt{update}_p\right\}$, the minor predecessor set $P_t = \texttt{premin}_t(c)$ is computed, and for each $p \in \bigcup_t P_t$, F is updated to be the set $F' = \texttt{min}(F\uparrow \cup p\uparrow)$. Any element no longer in F is removed from Q, and p is added to Q. F' is computed as follows:

$$F' = (F \setminus \{f \in F \mid p \sqsubseteq f\}) \cup \{p\}$$

The analysis terminates when either $p \in \texttt{Init}_{PWS}$ or Q is empty. If the algorithm terminates by finding a p s.t. $p \in \texttt{Init}_{PWS}$, some local state definition $\underline{q}$ in $forbidden$ is reachable, and if the algorithm terminates by Q becoming empty, no local state definition in $forbidden$ is reachable.

6.3 Optimisations

A problem with naive implementations of state exploration analysis algorithms like this one is the state explosion problem; that each variable (any state, not necessarily a memory location) introduces a factor on the number of possible states in the state space. Memorax uses four different approaches in order to reduce the size of the state space.

The first approach is explained in section 6.1. The constraint encoding allows for exploring all possible values of a variable at once, and only breaking the constraint into configurations with explicit values for that variable once it's value makes a difference in regards to reachability.

The second approach is to not consider $t = \texttt{serialise}_{p,x}$ unless $\underline{q}(p)$ is reachable using a write transition; i.e. the program state is just after a write. This works because waiting to do a serialise (excluding past another write) will not allow reaching more local states, but it increases the state space because the same trace except for the positions of the serialisations is explored multiple times. Neither is $t = \texttt{update}_p$ considered unless either there is a read (or atomic read-write) transition leaving $\underline{q}(p)$ or $\underline{z}(p) = |mem|$. This also limits the number of equivalent traces that can be used to reach a certain state.

The third approach is to perform a over-approximate forwards analysis of each automaton, and then use the information gathered to discard constraints that were never visited by the forwards analysis. The information gathered is which *last write sequences* that are possible for each process in each state. A last write sequence is a sequence of writes (pairs of memory locations and values), for example `z=1, |, y=3, x=2`, with exactly one occurrence of the symbol `|` and each write is the last write of that process to that memory location. I will use the more compact syntax `z=1|y=3:x=2` from now on. The symbol `|` follows the last of the writes that is a channel message $mem\,(i)$ such that $\underline{z}\,(p) \geq i$. The writes are ordered in memory order, that is, the order they will eventually have in the channel. Because the order they are written isn't necessarily memory order (if it was, we'd be dealing with TSO), an alternate representation, called a *last write set*, using partial functions (maps) from memory locations to values is used. The set `{z=1}|{y=3, x=2}` means that the writes to `y` and `x` may be arbitrarily ordered, but the write to `z` must be strictly earlier in memory order. I will also let sequences where the symbol `|` is later than in the set representation to be in the set. For example, both `z=1:y=3|x=2` and `z=1|x=2:y=3` are contained in the set `{z=1}|{y=3, x=2}`, but `|z=1:y=3:x=2` isn't.

Now, when performing the backwards reachability analysis, constraints where the last write sequence of some process isn't a member of any of the last write sets of that process' state can be excluded, as long as one of the last write that isn't matching is the last write that the process is observing. If the process is observing a later write by another process, then the last write of that process isn't necessarily the same in every configuration in the upwards closure (the converse is guaranteed by condition 4 of $\sqsubseteq$). Also, if the constraint contains the value $*$ as the last write, but all last write sets agree on a constant value, the constraint can be limited to that constant.

The fourth approach Memorax uses to reduce the size of the state space is based on the observation that if for a memory location x, $\forall p.\, \underline{b}\,(p)\,(x) = \varepsilon$, $mem\,(i)$ is the last message in mem to write to x, then for all $j \geq i$, $m'\,(x) = m\,(x)$ for $mem\,(i) = (m,\, p,\, x)\,,\, mem\,(j) = (m',\, q,\, y)$ in order for the configuration to be reachable. And since it also will be true for $i' = h\,(i)$ for any configuration in the upwards closure of the configuration, any constraint violating this condition can be discarded. If only a subset of configurations represented by a constraint does not violate this rule, the constraint can be substituted by another constraint representing that subset.

7 Performance evaluation

In this section I will compare the performance of the Memorax implementation of PWS with the Memorax implementation of SB.

In Table 1, we can see how the implementations of PWS and SB compare on some example algorithms (the source of these algorithms are distributed along with Memorax in the `doc/examples` folder). The "No. constraints" column contains the number of constraints that were visited (popped from Q) during the analysis. The time measurements were taken on a server with 128GB of RAM and AMD Bulldozer processors with 16K/2M/6M cache running at a maximum frequency of 3.6GHz. Memorax was compiled using Clang 3.3, optimisation level 2. The "Result" column is included because both implementations take more time and explore more constraints when when concluding that the forbidden states are unreachable, thus the performance is not directly comparable when the results differ. The only case where this might have happened is in the case

Algorithm	Abs.	Time	No. constraints	Result
Lamport's bakery (bounded)	SB	0.8 s	32220	Yes
	PWS	3.4 s	144022	Yes
Burns algorithm	SB	0.0 s	246	Yes
	PWS	0.0 s	375	Yes
CLH queue lock	SB	OOM	OOM	OOM
	PWS	964.7s	13715535	Yes
Dekkers lock with deadlock protection	SB	0.1 s	2540	Yes
	PWS	0.6 s	25944	Yes
Dijkstras lock	SB	0.2 s	9567	Yes
	PWS	1.0 s	42763	Yes
Lamport Fast	SB	0.3 s	14614	Yes
	PWS	7.5 s	354139	Yes
Peterson	SB	0.6 s	14178	Yes
	PWS	1.1 s	52290	Yes
Sense Reversing Barrier	SB	0.2 s	4976	No
	PWS	2.7 s	41570	No
Increasing Sequence[2] (bounded to 10)	SB	0.2 s	5580	No
	PWS	2.9 s	44202	No

Table 1: Reachability performance compared to SB

of CLH, but since the SB reachability analysis is unable to complete using less than 100GB of memory[1], its result is unknown. All results are measured using the register free form transformation of the program automata.

In Table 2, we can see how the implementations of PWS and SB compare on the same example algorithms as above, but at the task of finding all minimal sets of fences that makes the algorithms safe, using a trivial extension of the fence insertion procedure of [2] to PSO. The "Max no. constraints" column contains the largest number of constraints that were visited during any of the reachability analyses run. The "Same fence set" column compares the fence sets computed by SB and PWS. If the fence sets are different then some of the machines that were analysed will have differed, and the result must be interpreted with that in mind.

Note that for Lamports bakery, the reduction to register free form splits a line in the code into several transitions. In this case, PSO needs fences on all of the transitions, but one is sufficient for TSO. If the analysis wasn't done on register free automata, the fence sets would be the same.

8 Conclusion

I have presented a sound and complete approach to reachability analysis under PSO. This approach is based on the theory of well-structured systems and is applicable to any program with finite data domains. I have also presented an implementation of my approach as part of the Memorax tool, which is able to

[1]At the time this benchmark was performed, the default implementation had a bug that caused a large part of the state space to not be explored. The result I present here is using a crude workaround of the bug.

Algorithm	Abs.	Time	Max no. constraints	Same fence set
Lamports bakery (bounded)	SB	48.6 s	454109	No (see p. 19)
	PWS	41.5 s	287304	
Burns algorithm	SB	0.0 s	246	Yes
	PWS	0.0 s	375	
CLH queue lock	SB	OOM	OOM	n/a
	PWS	OOM	OOM	
Dekkers lock with deadlock protection	SB	0.1 s	2536	Yes
	PWS	2.2 s	25944	
Dijkstras lock	SB	0.4 s	6006	Yes
	PWS	6.2 s	59504	
Lamport Fast	SB	174.4 s	1255435	No
	PWS	308.4 s	2594464	
Peterson	SB	0.8 s	14199	No
	PWS	4.9 s	62055	

Table 2: Fence insertion performance compared to SB

analyse reachability and, using a trivial extension of the existing fence inference procedure, infer all minimal fence sets on typical non-trivial examples with reasonable time and memory usage.

However, I have not presented a proof to Conjecture 1, which, although not a far-fetched extension of Theorem 1 in [2], is critical to the correctness of my approach. Also, although my formalisation of PSO is based on the SPARC manual, there is no guarantee that the behaviour of actual SPARC processors in PSO mode is limited to what is possible in my formalisation. However, should SPARC, or some other processor with store-load and store-store reordering, turn out to be more allowing or significantly more strict than my formalisation, it is relatively easy to modify my formalisation, PWS, and Memorax to reflect that.

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 313–321. IEEE, 1996.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under tso. In *TACAS*, 2012.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under tso. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 530–536. Springer, 2013.

[4] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[5] Jade Alglave and Luc Maranget. Stability in weak memory models. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806, pages 50–66. Springer, Heidelberg, 2011.

[6] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. *SIGPLAN Not.*, 45(1):7–18, January 2010.

[7] Jacob Burnim, Koushik Sen, and Christos Stergiou. Testing concurrent programs on relaxed memory models. *UCB/EECS*, 32, 2010.

[8] Sen K. Burnim, J. and C Stergiou. Sound and completemonitoring of sequential consistency for relaxed memory models. In P.A. Abdulla and K.R.M Leino, editors, *TACTAS 2011*, volume 6605 of *LNCS*, pages 11–25. Springer, Heidelberg, 2011.

[9] Keir Fraser. *Practical lock-freedom.* PhD thesis, PhD thesis, Cambridge University Computer Laboratory, 2003., 2004. Also available as Technical Report UCAM-CL-TR-579.

[10] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. *Performance evaluation of memory consistency models for shared-memory multiprocessors*, volume 19. ACM, 1991.

[11] Vojtěch Havel. Relaxed memory models in DiVinE. 2012.

[12] G. Higman. Ordering by divisibility in abstract algebras. In *Proc. London Math. Soc.*, pages 326–336, 1952.

[13] Thuan Quang Huynh and Abhik Roychoudhury. A memory model sensitive checker for C#. In Nipkow T. Misra, J. and G Karakostas, editors, *FM 2006: Formal Methods*, volume 4085 of *LNCS*, pages 476–491. Springer, Heidelberg, 2006.

[14] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 111–119. IEEE, 2010.

[15] Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In *ACM SIGPLAN Notices*, volume 46, pages 187–198. ACM, 2011.

[16] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in pso memory systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 339–353. Springer, 2013.

[17] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics*, pages 391–407. Springer, 2009.

[18] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9.* PTR Prentice Hall, 1994. Chapter 8, Memory Models; p.119-132.