TSO Reachability 1 2
TSO Reachability 2 12

8. TSO Reachability

<u>Goal</u>: Solve reachability for parallel programs (assembly) running on x86 Total Store Ordering (TSO) architectures

<u>Approach</u>: Reduce to LCS reachability

8.1 Syntax of Parallel Programs

<u>Definition:</u>

We consider parallel programs with shared memory as defined by following grammar:

```
<prog> ::= program <pid> <thrd>*     // Name & finite number of threads
<thrd> ::= thread <tid>              // Identifier
             regs <reg>*             // List of local registers
             init <label>            // Initial instruction
             begin <linst>* end      // List of labelled instructions

<linst> ::= <label>: <inst>; goto <label>;
<inst> ::=  <reg> ← mem[<reg>]       // Load
          | mem[<reg>] ← <reg>       // Store
          | mfence                   // Memory fence
          | <reg> ← <expr>           // Local assignment
          | assert <expr>
<expr> ::= fun (<reg>*)
```

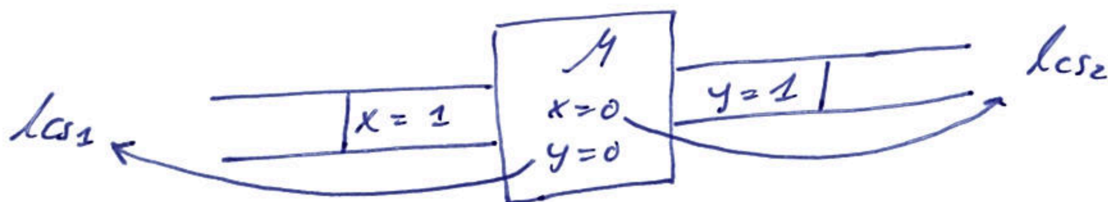
- We assume programs come with a <u>finite</u> <u>data domain</u> DOM and a <u>function</u> <u>domain</u> FUN that are
  - defined on DOM
  - all computable.
- We assume
  - $0 \in DOM$
  - All values in DOM can be used as addresses.

8.2 TSO-Semantics of Parallel Programs:

Idea of TSO:
- Each thread has a store buffer to reduce latency of memory accesses
- Stores are buffered locally and later propagated to the memory in batch processing style.
- To provide the illusion that a sequential program runs on a coherent memory, buffered stores are forwarded to later loads $\Rightarrow$ early reads

Example: Dekker's mutex

l_1: mem[x] $\leftarrow$ 1; goto l_2	l_1': mem[y] $\leftarrow$ 1; goto l_2'
l_2: r $\leftarrow$ mem[y]; goto l_3	l_2': r' $\leftarrow$ mem[x]; goto l_3'
l_3: assert(r=0); goto l_{cs_1}	l_3': assert(r'=0); goto l_{cs_2}
l_{cs_1}: critical section	l_{cs_2}: critical section



Fences instructions (mfence) stop a thread until all preceding writes have arrived in memory.

For the definition, fix a program P with threads $\{t_1, \dots, t_n\}$.
Assume t_i has identifier i,
initial label $l_{0,i}$,
and declares registers $\overline{r_i}$.

We use $TID := \{1, \dots, n\}$ for the thread identifiers,
LAB for the locations in all threads, and
$VAR := DOM \cup \bigcup_{i=1}^{n} \overline{r_i}$ for the addresses and registers.

The TSO semantics is <u>operational</u>,
in terms of transitions between configurations.

The set of <u>TSO-configurations</u> is

$$CF := LAB^{TID} \times DOM^{VAR} \times (DOM \times DOM)^{*\,TID}.$$

We write a <u>configuration</u> as

$$cf = (pc, val, buf),$$

where
- $pc(t)$ is the <u>program counter</u> of thread t.
- $val(r)$ is the <u>valuation</u> of register r, and
- $buf(t)$ is the <u>buffer content</u> for thread t, a sequence of address-value pairs.

The <u>initial configuration</u> is

$$cf_0 := (pc_0, val_0, buf_0)$$

with
$pc_0(t) = l_{0,t}$ for all $t \in TID$
$val_0(x) = 0$ for all $x \in VAR$
$buf_0(t) = \varepsilon$ for all $t \in TID$.

The <u>TSO-transition relation</u> $\rightarrow_{TSO} \subseteq CF \times CF$
is the smallest relation that satisfies the following rules,
where we assume $pc(t) = l$, an instruction $l: \langle inst \rangle;$ <u>goto</u> l';
and where we set $pc' := pc[t := l']$:

(EARLY)
$$\frac{\langle inst \rangle = r \leftarrow mem[r'],\ a = val(r') \qquad buf(t)\downarrow(a = *) = (a = v).\beta}{(pc, val, buf) \rightarrow_{TSO} (pc', val[r := v], buf)}$$

(LOAD)
$$\frac{\langle inst \rangle = r \leftarrow mem[r'],\ a = val(r'),\ v = val(a) \qquad buf(t)\downarrow(a = *) = \varepsilon}{(pc, val, buf) \rightarrow_{TSO} (pc', val[r := v], buf)}$$

$$(\text{STORE})\quad \frac{\langle inst\rangle = \text{mem}[r] \leftarrow r',\ a = val(r),\ v = val(r')}{(pc, val, buf) \rightarrow_{TSO} (pc', val, buf[t := (a,v).buf(t)])}$$

$$(\text{UPDATE})\quad \frac{buf(t) = \beta.(a,v)}{(pc, val, buf) \rightarrow_{TSO} (pc, val[a := v], buf[t := \beta])}$$

$$(\text{FENCE})\quad \frac{\langle inst\rangle = \underline{\text{mfence}},\ buf(t) = \varepsilon}{(pc, val, buf) \rightarrow_{TSO} (pc', val, buf)}$$

+ 2 rules for assignments and asserts.

8.3 From TSO to LCS

Goal: Represent TSO semantics of program P by LCS L_P.

Approach: Define LCS L_P^1, L_P^2, L_P^3, L_P with more features that fix problems in earlier versions.

Towards L_P^1:

Idea: · Shared memory communication is like message loss:
↳ a store may get overwritten before it is received by another thread.
· Understand TSO buffers as lossy channels

Problem: · Higman's ordering is no simulation relation for TSO.
↳ Consider

$$\begin{array}{ll} l_0 : ld(x,1); & l_0' : st(y,1); \\ l_1 : ld(y,0); \quad \| & l_1' : st(x,1); \\ l_2 & l_2' \end{array}$$

where $ld(x,1)$ is a shorthand for
$r \leftarrow mem[x];$
$assert(r=1);$

↳ Then configuration
$(l_0, l_2', x=0=y, \varepsilon, x=1.y=1)$
cannot reach l_2, l_2' $\quad \preceq$
whereas
$(l_2, l_2', x=0=y, \varepsilon, x=1)$ can.

| Lossiness gives inconsistent memory configurations. |

Fix: Let threads send entire memory snapshots
↳ Each message in a buffer defines values of all variables.
↳ The above problem vanishes.

$(l_0, l_2', x=0=y, \varepsilon, \begin{pmatrix} x=1 \\ y=1 \end{pmatrix}\begin{pmatrix} x=0 \\ y=1 \end{pmatrix})$

and

$(l_0, l_2', x=0=y, \varepsilon, \begin{pmatrix} x=1 \\ y=0 \end{pmatrix})$ are incomparable.

Towards L_P^2:

Problem: · Some behaviour under L_P^1 is not possible under TSO.

Example:

l_0: st(y, 0); ‖ l_0': st(x, 1)
l_1: ‖ l_1': ld(x, 0)
‖ l_2':

↳ Then (l_1, l_2') is not TSO-reachable.
TSO can only load 1 from x after store has been performed.

↳ The configuration is reachable in L_P^1.

$(l_0, l_0', x=0=y, \varepsilon, \varepsilon)$

$\to^2 (l_1, l_1', x=0=y, \begin{pmatrix} x=0 \\ y=0 \end{pmatrix}, \begin{pmatrix} x=1 \\ y=0 \end{pmatrix})$

$\to (l_1, l_1', \begin{matrix} x=1, \\ y=0 \end{matrix} \begin{pmatrix} x=0 \\ y=0 \end{pmatrix}, \varepsilon)$

$\to (l_1, l_1', x=0=y, \varepsilon, \varepsilon)$

$\to (l_1, l_2', x=0=y, \varepsilon, \varepsilon)$.

> Threads do not synchronize on memory updates
> (threads may use values that are no longer in memory).

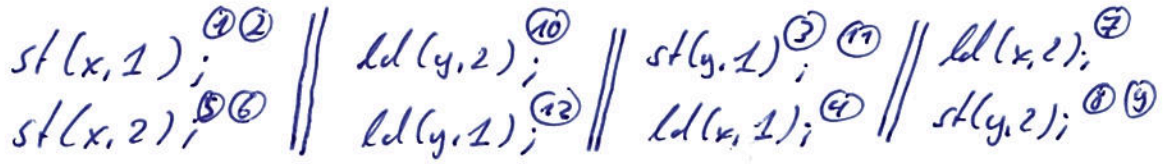
Fix: · Let all threads share the same buffer.

$\begin{pmatrix} x=1 \\ y=0 \end{pmatrix} . \begin{pmatrix} x=1 \\ y=0 \end{pmatrix}$ Store y = 0 takes x = 1 into account.

Towards L_P^3:

Problem: Some TSO behaviour is not possible in L_P^2.

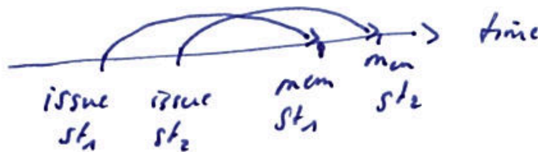
Example:



↳ This behaviour is not possible on L_P^2 for the following reason:

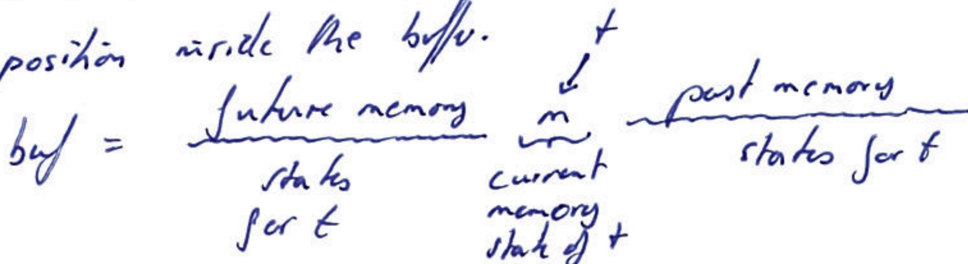
- st(y,1) has to be sent to memory before st(y,2)
  - ↳ Otherwise value of x will be (and stay) 2.
  - ↳ Then ld(x,1) is blocked.
- In L_P^2, operations in the buffer will be delivered to the memory in the order in which they entered the buffer.
  - ↳ Then thread t_2 is not able to load y=2 before y=1.

L_P^2 forces memory updates to occur in the same order as the order of the corresponding stores.

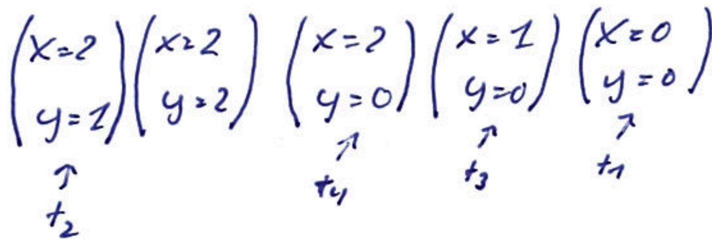


In TSO, the memory updates can be performed in opposite order, as the stores stem from different buffers.

Fix: Add to each thread a pointer to a position inside the buffer.

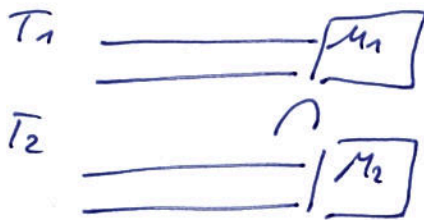


With this, updates are simulated by moving the pointer to the left.



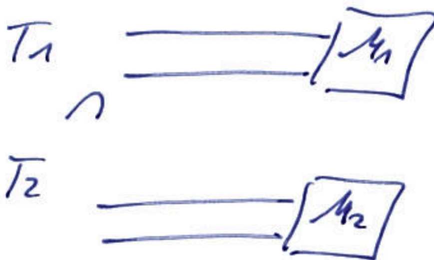
Why is this correct?

- TSO can be understood as



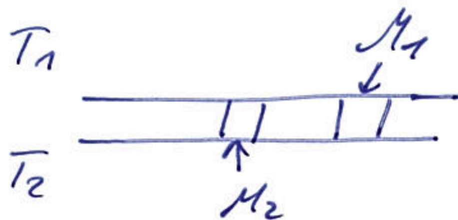
The automata synchronize when updates leave the channels.
↳ The update is performed on M_1 and M_2.

- Since the channels are FIFO, this is equivalent with



Here, the channels have the same content (one channel) but move to the memory at different speed.

- Since the channel content is identical, this is equivalent with



This is Lp^3.

Towards Lp:

Problem: In Lp^3, early reads are still missing.

Fix: Remember the last write of a thread to an address.

$\begin{pmatrix} x=2 \\ y=1 \end{pmatrix}, (t_3, y)) . (\begin{matrix} x=2 \\ y=2 \end{matrix}, (t_4, y)) . (\begin{matrix} x=2 \\ y=0 \end{matrix}, (t_1, x)) . (\begin{matrix} x=1 \\ y=0 \end{matrix}, /) . (\begin{matrix} x=0 \\ y=0 \end{matrix}, /)$

↑ t_2 ↑ t_4 ↑ t_3 ↑ t_1

<u>Construction:</u>

Given a program P,
we define the LCS with <u>strong symbols</u>

$L_P = (Q, q_0, C, M, S, \rightarrow)$ ...... strong symbols = cannot be lost, but bounded.

where

$Q := LOC^{TID} \times DOM^{VAR}$

$C := \{buf\}$

$M := DOM^{DOM}$

$S := \left(DOM^{DOM} \times (TID \times DOM \cup \{\varepsilon\}) \times \mathbb{P}(TID)\right) \setminus \left(DOM^{DOM} \times \{\varepsilon\} \times \{\emptyset\}\right)$

In a configuration (pc, val),

pc = current program counter

val = valuation of local registers, copy of last memory valuation that was sent to the buffer

To define the transitions,
consider $(pc, val) \in Q$ with $pc(t) = l$.

<u>Store:</u> If $l: mem[r] \leftarrow r'$; goto l' and $val(r) = a$, $val(r') = v$,
then we have the following LCS transition:

$$(pc, val) \xrightarrow[buf\,!\,(val[a:=v],\,(a,t),\,\emptyset)]{buf\,\{(mval,\,\varepsilon,\,tidset)\,/\,(mval,\,(t,a),\,tidset)\}} (pc[t:=l'], val[a:=v]).$$

(restricted to DOM)

Load: If $l: r \leftarrow mem[r']$; goto l'
and $val(r') = a$, there are two transitions:

$(pc, val) \xrightarrow{\underline{assert}\ (mval, (t,a), tidset) \in buf} (pc[t := l'], val[r := mval(a)])$

// early read

$(pc, val) \xrightarrow[\text{and } (mval, *, \{t\} \cup tidset) \in buf]{\underline{assert}\ (mval', (t,a), tidset') \notin buf} (pc[t := l'], val[r := mval(a)])$

Fence: If l: mfence; goto l', then

$(pc, val) \xrightarrow{\underline{assert}\ (mval, *, \{t\} \cup tidset) = head(buf)} (pc[t := l'], val)$.

Update:

$(pc, val) \xrightarrow{\underline{assert}\ buf = w_1.m_1.m_2.w_2 \quad \underline{modify}\ buf = w_1.m_1'.m_2'.w_2} (pc, val)$

where
$m_1 = (mval_1, last_1, tidset_1)$
$m_2 = (mval_2, last_2, tidset_2 \cup \{t\})$
$m_1' = (mval_1, last_1 \setminus \{(t, *)\}, tidset_1 \cup \{t\})$
$m_2' = (mval_2, last_2, tidset_2)$.

Theorem (Atig, Bouajjani, Burckhardt, Musuvathi 2010'12'14):
Consider a program P.
A control-state pc is reachable when P is executed under TSO
iff pc is reachable in L_P.
Since the latter problem is decidable (Abdulla 1996),
control-state reachability is decidable for TSO.

Intuition to the construction from last time:

The construction makes heavy use of the following trick in automata theory:

to shuffle two languages $\mathcal{L}(A_1)$ and $\mathcal{L}(A_2)$ over disjoint alphabets $\Sigma_1 \cap \Sigma_2 = \emptyset$,
extend A_1 by loops Σ_2 on each state,
and A_2 by loops Σ_1.
The resulting automata A_1' and A_2' satisfy

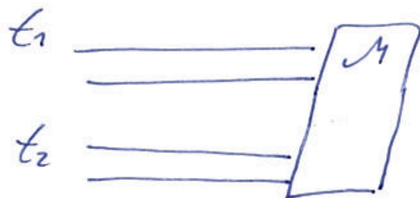
$$\mathcal{L}(A_1') \cap \mathcal{L}(A_2') = \mathcal{L}(A_1) ш \mathcal{L}(A_2).$$

Example:

$\{ab\}$ ш $\{xyz\}$

$= \mathcal{L}(\to q \xrightarrow{a} q \xrightarrow{b} q_f$ with loops x,y,z on each state$) \cap \mathcal{L}(\to q \xrightarrow{x} q \xrightarrow{y} q \xrightarrow{z} q_f$ with loops a,b on each state$)$.

1.) The memory sees a shuffle of the stores from both threads:

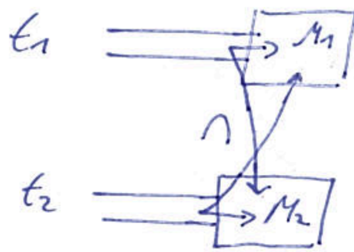


For example, the memory may see

$$\rho = st(a,1,t_1).\, st(b,2,t_2).\, st(a,2,t_1).$$

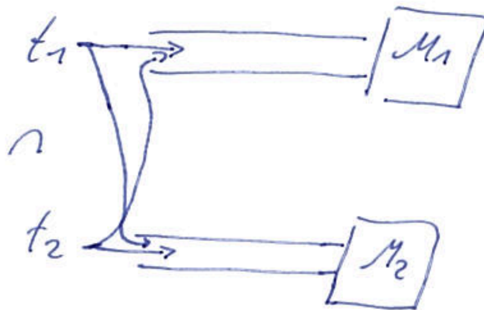
2.) Using the above idea, the shuffle can be obtained by synchronizing (intersection) the two automata $t_1 = M_1$ and $t_2 = M_2$ on the memory updates:

- 1 -

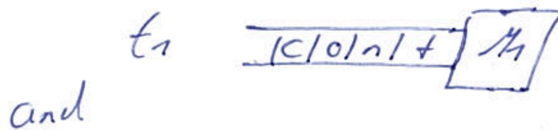


Here, one really defines alphabets (Σ_i consists of stores $st(a, v, t_i)$, $i = 1, 2$), and applies the loop trick.

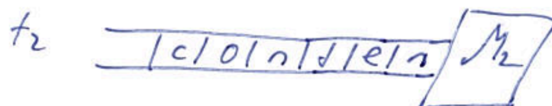
3.) Since the channels are FIFO, the stores leave the buffer in the order in which they are put into the buffer.
So instead of guessing the buffered stores of t_2 <u>at the memory</u>,
thread t_1 guesses the stores of t_2 already <u>when issuing</u> commands into the buffer:



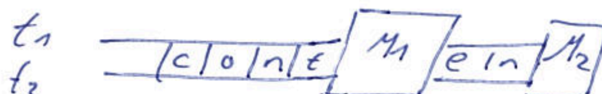
4.) Now the content of both buffers is identical, up to the fact that M_1 and M_2 process the buffer at different speed:



and



can be understood as one buffer with two pointers:



Remark:

One may think that it should be sufficient to guess the sequence of memory updates

$\rho = st(a,1,t_1).st(b,2,t_2).st(a,2,t_1)$

and write this shared sequence into both buffers:

This, however, yields a strict under-approximation of TSO.

TSO requires that sequence ρ results from store actions leaving the buffer.

The above model additionally enforces this order when the stores are send to the buffer.

↳ This is more than TSO asks for.

↳ That the two memories can process the shared buffer at different speed fixes the problem.