8. TSO Reachability

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

Approach: Reduce to SCS reachability

8.1 Syntax of Parallel Programs

Definition:

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>*)
```

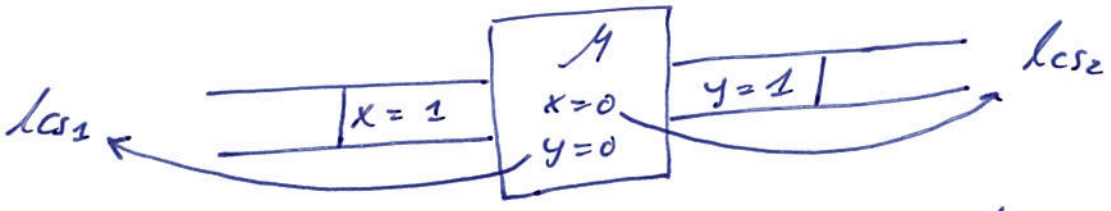
- We assume programs come with a finite data domain DOM and a function domain FUN that are
  - defined on DOM
  - all computable.
- We assume
  - $0 \in DOM$
  - All values in DOM can be used as addresses.

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
  ⇒ early reads

Example: Dekker's mutex

l_1: mem[x] ← 1; goto l_2	l_1': mem[y] ← 1; goto l_2'
l_2: r ← mem[y]; goto l_3	l_2': r' ← 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 operational,
in terms of transitions between configurations.

The set of TSO-configurations is

$$CF := \mathcal{L}\mathit{AB}^{TID} \times DOM^{VAR} \times (DOM \times DOM)^{*\,TID}.$$

We write a configuration as

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

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

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

$$(\text{FLUSH}) \quad \frac{\langle inst \rangle = r \leftarrow mem[r'],\ a = val(r') \qquad buf(t)\downarrow(a = *) = (a = v) \cdot \beta}{(pc, val, buf) \to_{TSO} (pc', val[r := v], buf')}$$

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

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

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

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

+ 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

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

where ld(x, 1) is a shorthand for

r ← 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' ↓ ⇏ ↓

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 store 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 LP^2:

Problem: · Some behaviour under LP^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 LP^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.

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

Towards LP^3:

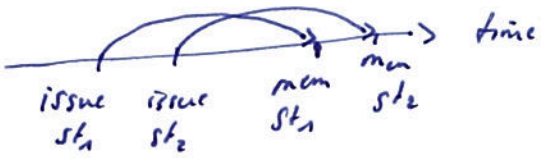
Problem: Some TSO behaviour is not possible in LP^2.

Example:

$st(x,1)$; ① ②	$ld(y,2)$; ⑩	$st(y,1)$; ③ ⑪	$ld(x,2)$; ⑦
$st(x,2)$; ⑤ ⑥	$ld(y,1)$; ⑫	$ld(x,1)$; ④	$st(y,2)$; ⑧ ⑨

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

- $st(y,2)$ 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 LP^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 $y=1$.

> LP^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.

$$buf = \underbrace{\text{future memory states for } t}_{\text{current memory state of } t} \overset{\downarrow t}{\underbrace{m}} \underbrace{\text{past memory states for } t}$$

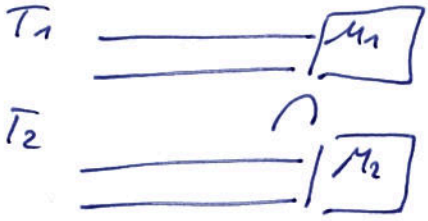
With TSO, updates are simulated by moving the pointer to the left.

$\begin{pmatrix} x=2 \\ y=1 \end{pmatrix}$ $\begin{pmatrix} x=2 \\ y=2 \end{pmatrix}$ $\begin{pmatrix} x=2 \\ y=0 \end{pmatrix}$ $\begin{pmatrix} x=1 \\ y=0 \end{pmatrix}$ $\begin{pmatrix} x=0 \\ y=0 \end{pmatrix}$

t_2 t_4 t_3 t_1

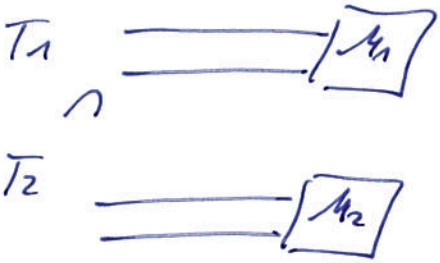
Why is TSO correct?

- TSO can be understood as



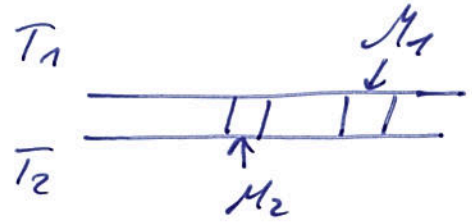
The automata synchronize when updates leave the channels.
↳ The update is performed on M_1 and M_2.

- 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.

$\left(\begin{matrix}x=2\\y=1\end{matrix}, (t_3, y)\right) \Big| \left(\begin{matrix}x=2\\y=2\end{matrix}, (t_4, y)\right) \Big| \left(\begin{matrix}x=2\\y=0\end{matrix}, (t_1, x)\right) \Big| \left(\begin{matrix}x=1\\y=0\end{matrix}, \checkmark\right) \Big| \left(\begin{matrix}x=0\\y=0\end{matrix}, \checkmark\right)$

t_2 t_4 t_3 t_2

Construction:

Given a program P,
we define the LCS with strong symbols

$$L_P = (Q, q_0, C, M, S, \to)$$

..... 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 $pc(t) = l$.

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

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

restricted to Dom

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

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

// early read

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

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

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

Update:

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

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

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 LP.
Since the latter problem is decidable (Abdulla 1996),
control-state reachability is decidable for TSO.