10. Robustness against Total Store Ordering

(Bouajjani, M., Mühlmann, ICFLP 2011)
(Bouajjani, Derevenetc, M., ESOP 2013)

Claim:
- Programmers think in terms of sequential consistency
- Hence, behaviour that deviates from SC should be considered a programming error.

Correctness criterion:
- The program behaviour under TSO should coincide with the SC behaviour:

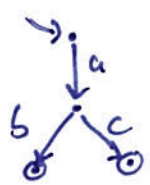
$$B_{TSO}(P) = B_{SC}(P).$$

- In this case, the program is called robust against (execution under) TSO.

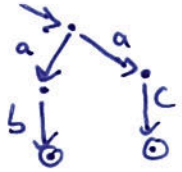
Notion of behaviour:

Trade-off:
↳ Want a weak comparison between TSO and SC to maximise the benefits of relaxed executions
↳ The weaker the correspondence between TSO and SC, the harder it is to check the correspondence.

(Analogue:
Language equivalence $=_L$ between finite automata is PSPACE-complete
while
the stronger bisimulation equivalence $=_B$ is in PTIME



$=_L$
$\neq_B$



)

Options:

- State-based robustness: $Reach_{TSO}(P) = Reach_{SC}(P)$
  - ↳ very good: weak notion that allows for quite relaxed executions.
  - ↳ bad: as hard to check as reachability under TSO.

- (Happens-before) Trace-based robustness: $Traces_{TSO}(P) = Traces_{SC}(P)$
  - ↳ good: still quite relaxed executions possible
  - ↳ very good: only PSPACE-complete.

Goal: (1) • Show that trace-based robustness is decidable, actually PSPACE-complete.
• Reduce trace-based robustness to classic SC reachability.

(2) Develop an algorithm that turns a non-robust program into a robust one.

10.1 Traces and Robustness

- To define robustness, we define (SC-) happens-before traces
- To define happens-before traces, we define computations under SC and under TSO.
- To define computations, we label the transition relation by actions from

$$ACT := TID \times (\{isu, loc\} \cup \{ld, st\} \times DOM \times DOM)$$

Definition:
The labelled TSO transition relation

$$\rightarrow_{TSO} \subseteq CF \times ACT \times CF$$

is defined by the following rules, which assume $cf = (pc, val, buf)$ with $pc(t) = l$ with $l: \langle inst \rangle$ goto l' and $pc' := pc[t := l']$:

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

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

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

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

The remaining transitions are labelled by (t, loc).

The set of TSO computations is

$$C_{TSO}(P) := \{\ \tau \in ACT^* \mid s_0 \xrightarrow{\tau}_{TSO} s \text{ for some } s = (pc, val, buf) \text{ with } buf(t) = \varepsilon \text{ for all } t \in TID\ \}$$

For sequential consistency (SC), stores are not buffered – to be precise, buffered and immediately flushed.

So $C_{SC}(P)$ is a special case of the TSO computations.

Example (Dekker = Store buffering (SB))

l_0 : mem[x] ← 1 ; goto l_1 ; ‖ l_0' : mem[y] ← 1 ; goto l_1' ;
l_1 : r_1 ← mem[y] ; goto l_2 ; ‖ l_1' : r_2 ← mem[x] ; goto l_2' ;

If TSO computation:

$\tilde{\tau} = (t_1, isu).(t_1, ld, y, 0).(t_2, isu).(t_2, st, y, 1).(t_2, ld, x, 0).(t_1, st, x, 1)$

Definition (Trace):

Consider $\tilde{\tau} \in C_{TSO}(P)$.
Its trace $Tr(\tilde{\tau})$ is a node-labelled graph

$Tr(\tilde{\tau}) := (N, \lambda, \rightarrow_{po}, \rightarrow_{sl}, \rightarrow_{src})$

with
- set of nodes N
- labelling $\lambda : N \to ACT$
- $\rightarrow_{po}, \rightarrow_{sl}, \rightarrow_{src} \subseteq N \times N$

  program-order, store-order (coherence relation), and source-relation (reads-from)

The definition is by induction on the length of the computation:

↳ Empty word ε ⇝ empty trace

↳ Assume $Tr(\tilde{\tau}) = (N, \lambda, \rightarrow_{po}, \rightarrow_{st}, \rightarrow_{src})$.

Then $Tr(\tilde{\tau}.act) := (N \cup \{n\}, \lambda', \rightarrow'_{po}, \rightarrow'_{st}, \rightarrow'_{src})$,

where the choice of node n depends on the type of act:

act = (t, st, a, v):
Let n be the minimal node in $\rightarrow_{po}^{t}$ labelled by $\lambda(n) = isu$.

Set $\lambda' := \lambda[n := act]$ and $\rightarrow_{po}' := \rightarrow_{po}$.

// If we have a store, we use the moment the action was issued.

$act \neq (t, st, a, v)$:

Add a fresh node $n \notin N$ to the trace.

Set $\lambda' := \lambda \cup \{(n, act)\}$,

$\rightarrow_{po}' := \rightarrow_{po} \cup \{(max(\rightarrow_{po}^{t}), n)\}$.

Store order $\rightarrow_{st}'$:

Only updated for stores (t, st, a, v):

$\rightarrow_{st}' := \rightarrow_{st} \cup \{(max(\rightarrow_{st}^{a}), n)\}$

Not changed otherwise.

Source relation $\rightarrow_{src}'$:

Only updated for loads and stores:

Load (t, ld, a, v):

$\rightarrow_{src}' := \rightarrow_{src} \cup \{(max(\rightarrow_{st}^{a}), n)\}$.

Store (t, st, a, v):

Update the source relation for all loads that read early from this store:

$\forall m \in N$ with $n \rightarrow_{po}^{+} m$ and $\lambda(m) = (t, ld, a, v)$:

$\rightarrow_{src}' := (\rightarrow_{src} \setminus \{(*, m)\}) \cup \{(n, m)\}$.

Consider $Tr(\tau) = (N, \lambda, \rightarrow_{po}, \rightarrow_{st}, \rightarrow_{src})$.

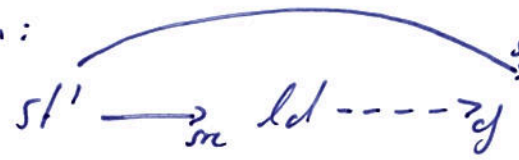
The conflict relation (from-read) is derived from $\rightarrow_{src}$ and $\rightarrow_{st}$.

We define

$ld \rightarrow_{cf} st$ if $\exists st'$: $st' \rightarrow_{src} ld$ and $st' \rightarrow_{st} st$

or ld loads the initial value and st is the first store on the address

Illustration:



The (SC-) happens-before relation of the trace is

$$\rightarrow_{hb} := \rightarrow_{po} \cup \rightarrow_{rf} \cup \rightarrow_{src} \cup \rightarrow_{cf}.$$

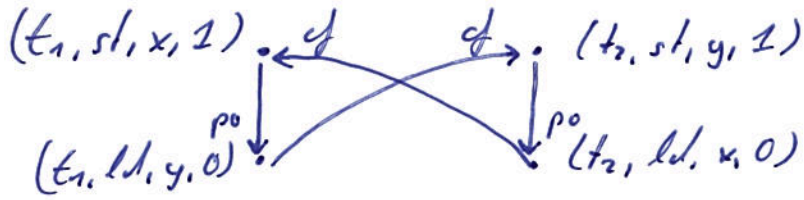
Let $Tr_{TSO/SC}(P) := Tr(C_{TSO/SC}(P))$ denote the set of all TSO/SC traces of P.

Example:

Consider

$$\tau = (t_1, isu).\ (t_1, ld, y, 0).\ (t_2, isu).\ (t_2, st, y, 1).\ (t_2, ld, x, 0)\ (t_1, st, x, 1.$$

The corresponding trace + conflict relation is



The decision problem we tackle is the following:

Definition (Robustness):

Given: A parallel program P.

Problem: Does $Tr_{TSO}(P) = Tr_{SC}(P)$ hold?

Note:

This notion of trace-based robustness is really stronger than state-based robustness:

Lemma:

If $Tr_{TSO}(P) = Tr_{SC}(P)$ then $Reach_{TSO}(P) = Reach_{SC}(P)$.

The reverse implication does not hold.

Proof: Homework.

Note:

↳ Inclusion $Tr_{SC}(P) \subseteq Tr_{TSO}(P)$ always holds,
have to check the reverse inclusion.

↳ How to check $Tr_{TSO}(P) \subseteq Tr_{SC}(P)$?
Cannot complement an automaton.

Characterisation of robustness:

↳ Computation $\bar{c} \in C_{TSO}(P)$ is called violating
if $Tr(\bar{c}) \notin Tr_{SC}(P)$.

↳ Observe that violating computations employ
cyclic accesses to addresses
that SC is unable to serialize.

↳ These cyclic accesses are made visible
by the conflict relation.

Lemma (Shasha & Snir, TOPLAS 1988):
Consider $Tr(\bar{c}) \in Tr_{TSO}(P)$.
Then $Tr(\bar{c}) \in Tr_{SC}(P)$ iff $\rightarrow_{hb}$ is acyclic.

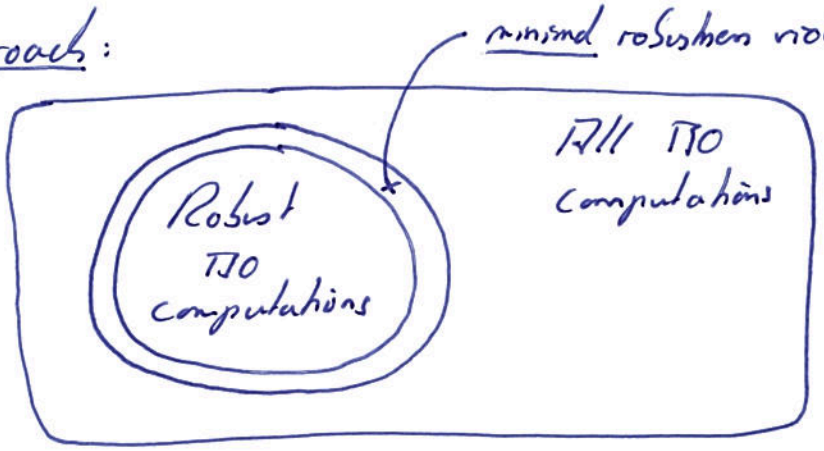
Proof:
„⇒" SC only generates computations
with acyclic hb-relation.
„⇐" Every partial order can be extended
to a total order.
This total order is an SC computation. □

Comment:

↳ Result of Shasha & Snir is semantical,
does not provide an algorithm to find cyclic traces.

↳ Our main goal is to show
that cyclic traces can be found in PSPACE.

Approach:



(1) Understand shape of minimal robustness violations
(locality result for TSO: only a single thread has to delay stores).

(2) Devise an algorithm to detect violations of this shape.

Combines
- combinatorial reasoning for (1)

with
- algorithm design for (2).