10. Robustness against Total Store Ordering

(Bouajjani, M., Möhlmann, CAV LP 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:

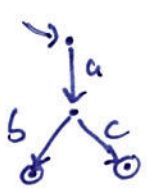
$$B_{TSO}(P) = B_{SC}(P).$$

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

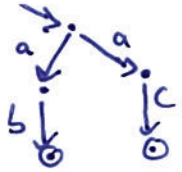
Notion of behaviour:

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

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

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 $pc' := pc[t := l']$:

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

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

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

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

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.

-2-

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:

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

Definition (Trace):

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

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

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

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

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

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

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

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

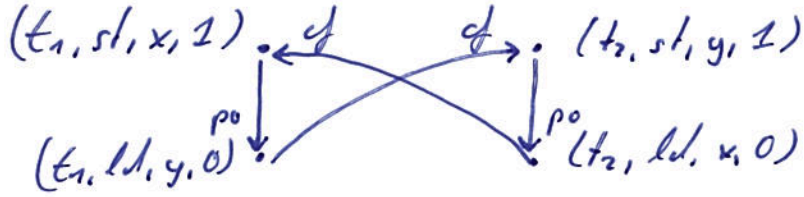
Let $Tr_{TSO/SC}(P) := Tr(C_{TSO/SC}(P))$ denote the set of all TSO/SC traces of P.

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.

-6- [100]: 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.

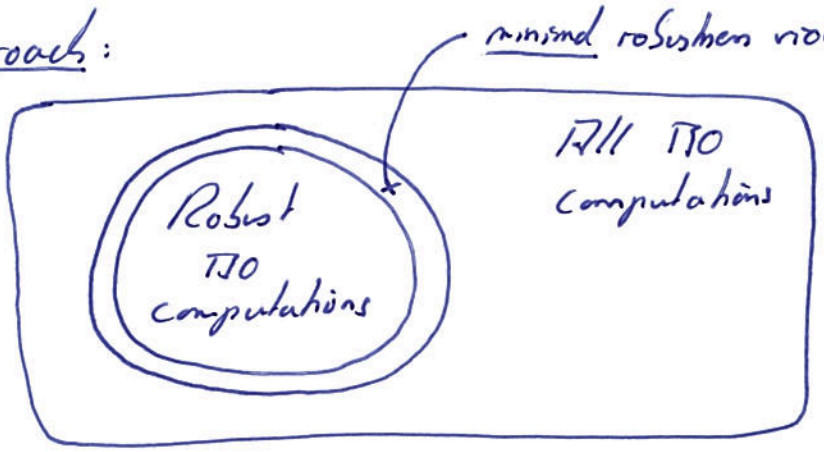
Proof:
„⇒" SC only generates computations with acyclic hb-relation.
„⇐" Every partial order can be extended to a total order.
This total order is an SC computation. □

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).

10.2 Minimal Violations and Locality:

Goal: Show that in a minimal violation only a single thread reorders its actions.

Definition (Minimal violation):

Consider $\tau = \alpha.a.\beta.b.\gamma \in C_{TSO}(P)$ with $thread(a) = t = thread(b)$.

- Then the distance of a and b in τ is $d_\tau(a,b) := |\beta \downarrow t|$.
- The number of delays in τ is

$$\#(\tau) := \sum_{\substack{\text{corresponding} \\ isu, st \in \tau}} d_\tau(isu, st).$$

- If violating computation τ is minimal if $\#(\tau)$ is minimal among all violating computations.

Note:
Program P is not robust iff it has a minimal violation.

Lemma 1 (Delays in minimal violations are required):

Consider $\tau = \alpha.isu.\beta.st.\gamma \in C_{TSO}(P)$ a minimal violation with isu, st from the same instruction of thread t.

Then $\beta \downarrow t = \varepsilon$
or $\beta \downarrow t = \beta'.ld.\beta''$ with $addr(ld) \neq addr(st)$ and β'' contains only stores.

Proof: Suppose β contains one or more actions of thread t.

- If all actions of t in β are stores, then also $\tau' := \alpha.\beta.isu.st.\gamma \in C_{TSO}(P)$.

The computation has the same trace as τ, but

$$\#(\tau') < \#(\tau). \quad \lightning \text{ Minimality.}$$

- Let a be the last non-store action in β of t:

$$\beta = \beta_1 . a . \beta_2 .$$

This means all actions of t in β_2 are stores, the remaining actions belong to other threads.

- Since store actions cannot be delayed past a fence, a is (1) issue, (2) local action, (3) or load.

In case (1), (2), and (3) with early load ($addr(a) = addr(st)$), delaying st past a can be avoided:

$$\tau' := \alpha . isu . \beta_1 . \beta_2 . st . a . \gamma \in C_{TSO}(P).$$

Again $Tr(\tau') = Tr(\tau)$ and $\#(\tau') < \#(\tau)$. $\lightning$ □

Goal: Detect happens-before cycles in a trace (graph structure) on the computation (linear structure).

Definition (Happens before through):

Let $\tau = \alpha . a . \beta . b . \gamma \in C_{TSO}(P)$.

Then a happens before b through β if there is a subsequence $c_1, \dots, c_n$ of β with

$$c_i \longrightarrow_{hb} c_{i+1} \quad \text{or} \quad c_i \longrightarrow_{po}^{*} c_{i+1} \quad \text{for } 0 \le i \le n$$

with $c_0 := a$ and $c_{n+1} := b$.

Lemma 2 (Happens before through is stable under insertion):

Consider $\tau = \alpha.a.\beta.b.\gamma \in C_{TSO}(P)$

and $\tau' = \alpha'.a.\beta'.b.\gamma'$

so that $\tau \downarrow t = \tau' \downarrow t$ for all $t \in TID$.

Moreover, assume β is a subsequence of β'.

If $a \to_{hb}^{+} b$ through β, then $a \to_{hb}^{+} b$ through β'.

Proposition (Dichotomy):

Consider a minimal violation $\tau = \alpha.a.\beta.b.\gamma \in C_{TSO}(P)$.

Then

(1) $a \to_{hb}^{*} b$ through β

or (2) $\exists\, \tau' = \alpha.\beta_1.b.a.\beta_2.\gamma \in C_{TSO}(P)$

so that $\tau_r(\tau') = \tau_r(\tau)$

and $\tau' \downarrow t = \tau \downarrow t$ for all $t \in TID$.

Proof:

Showing (1) v (2) is equivalent to $\neg(1) \Rightarrow (2)$,

which is what we prove.

We proceed by induction on the length of β

and strengthen the hypothesis:

We additionally show that β_2 is a subsequence of β.

IH: Then $\tau = \alpha.a.b.\gamma$ and $a \not\to_{hb} b$.

$|\beta| = 0$ • If thread(a) = thread(b), then $b \to_{po}^{+} a$.

Therefore, b is a store action which has been delayed past a.

Swapping a and b will save the delay

without changing the trace. ↯ minimality.

- If $thread(a) \neq thread(b)$, then either
  - ↳ one of the actions is local,
  - ↳ the actions access different addresses, or
  - ↳ both are loads.

  In all three cases, swapping the actions produces $\bar{\tau}'$ as required.

IS: Assume the statement holds for all β' with $|\beta'| \leq n$.
Consider $\bar{\tau} = \alpha.a.\beta.c.b.\gamma$ with $|\beta.c| = n+1$.

Since we assume $a \xrightarrow{+}_{hb} b$ through $\beta.c$,
we have $a \xrightarrow{+}_{hb} c$ through β or $c \xrightarrow{+}_{hb} b$.

Let $a \xrightarrow{+}_{hb} c$ through β:
We apply the induction hypothesis to a and c.
This gives $\bar{\tau}' = \alpha.\beta_1.c.a.\beta_2.b.\gamma$
with $Tr(\bar{\tau}') = Tr(\bar{\tau})$ and β_2 a subsequence of β.
and $\bar{\tau}'|_t = \bar{\tau}|_t$ for all $t \in TID$

If we had $a \xrightarrow{+}_{hb} b$ through β_2 in $\bar{\tau}'$,
then also $a \xrightarrow{+}_{hb} b$ through $\beta.c$ in $\bar{\tau}$.
This holds by the induction hypothesis
(that β_2 is a subsequence of β)
together with Lemma 2,
and contradicts the assumption $a \xrightarrow{+}_{hb} b$ through $\beta.c$.

Hence, we can apply the hypothesis to a and b in $\bar{\tau}'$.
This yields
$\bar{\tau}'' = \alpha.\beta_1.c.\beta_{21}.b.a.\beta_{22}.\gamma$.

Again
$Tr(\bar{\tau}'') = Tr(\bar{\tau}')$ and β_{22} a subsequence of β_2.
and $\bar{\tau}''\downarrow t = \bar{\tau}'\downarrow t$ for all $t \in TID$

Together:
- $Tr(\bar{\tau}'') = Tr(\bar{\tau})$
- $\bar{\tau}''\downarrow t = \bar{\tau}\downarrow t$ for all $t \in TID$
- β_{22} is a subsequence of β_2,
  which is a subsequence of β_1,
  which is a subsequence of $\beta.c$,
  so β_{22} is a subsequence of $\beta.c$.

Let $c \to_{hs} b$:

We apply the induction hypothesis to b and c.
This yields
$\bar{\tau}' = \alpha\, a\, \beta.b.c\, \gamma$
with $Tr(\bar{\tau}') = Tr(\bar{\tau})$ and $\bar{\tau}'\downarrow t = \bar{\tau}\downarrow t$ for all $t \in TID$.
We now apply the hypothesis to a and b in $\bar{\tau}'$
and get
$\bar{\tau}'' = \alpha.\beta_1.b\,a.\beta_2.c\,\gamma$
with $Tr(\bar{\tau}'') = Tr(\bar{\tau}')$, $\bar{\tau}''\downarrow t = \bar{\tau}'\downarrow t$ for all $t \in TID$,
and β_2 a subsequence of β.

Together, $Tr(\bar{\tau}'') = Tr(\bar{\tau})$, $\bar{\tau}''\downarrow t = \bar{\tau}\downarrow t$ for all $t \in TID$,
and $\beta_2.c$ is a subsequence of $\beta.c$.

□

Recapitulation:

Goal: Establish locality

In a minimal violation, only a single thread delays stores.

Tool: Happens-before through

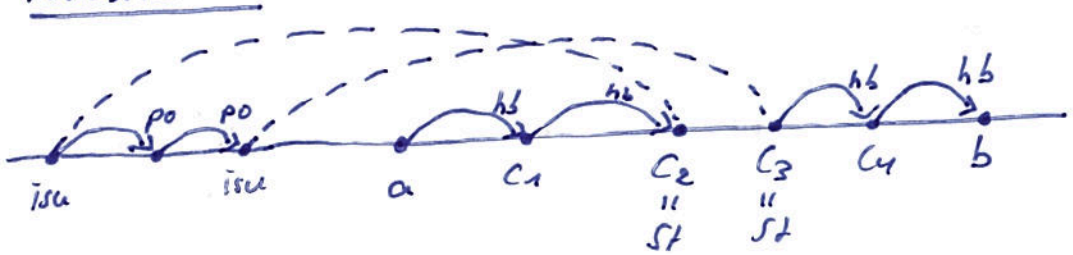
Consider $\bar{\tau}_1\, a\, \bar{\tau}\, b\, \bar{\tau}_2 \in C_{TSO}(P)$.

Then **a happens-before b through $\bar{\tau}$** if there is a subsequence $c_1, \dots, c_n$ of $\bar{\tau}$ so that

$$c_i \xrightarrow[\text{src, cf, st}]{} c_{i+1} \quad \text{or} \quad c_i \xrightarrow{+}_{po} c_{i+1}$$

for all $0 \le i \le n$ with $c_0 := a$ and $c_{n+1} := b$.

Illustration:



Proposition (Dichotomy):

Consider a minimal violation $\bar{\tau} = \bar{\tau}_1\, a\, \bar{\tau}_2\, b\, \bar{\tau}_3 \in C_{TSO}(P)$

Then

(1) $a \xrightarrow{+}_{hb} b$ through $\bar{\tau}_2$

or

(2) There is $\bar{\tau}' = \bar{\tau}_1\, \bar{\tau}_{21}\, b\, a\, \bar{\tau}_{22}\, \bar{\tau}_3 \in C_{TSO}(P)$

with $Tr(\bar{\tau}) = Tr(\bar{\tau}')$

and $\bar{\tau} \downarrow t = \bar{\tau}' \downarrow t$ for all $t \in TID$.

The proposition gives rise to the following principle for proving cycles:

"In a minimal violation, whenever a store is delayed past a load there is a happens-before cycle (that includes the two)."

Corollary:
Consider a minimal violation $\tau = \tau_1.\mathrm{isu}.\tau_2.\mathrm{ld}.\tau_3.\mathrm{st}.\tau_4 \in L_{TSO}(P)$, where st is the store corresponding to isu.
Then $Tr(\tau)$ is cyclic.

Proof:
We show $\mathrm{st} \xrightarrow{T}_{po} \mathrm{ld} \xrightarrow{+}_{hs} \mathrm{st}$.

- The same dependence holds since isu is issued before ld.
- For $\mathrm{ld} \xrightarrow{+}_{hs} \mathrm{st}$ we argue that $\mathrm{ld} \xrightarrow{+}_{hs} \mathrm{st}$ through τ_3.

Indeed, with dichotomy we have
(1) $\mathrm{ld} \xrightarrow{+}_{hs} \mathrm{st}$ through τ_3
or (2) a reordering of ld and st that does not change the trace and the threads' computations.

If we reordered ld and st using (2), then we would close $\tau \downarrow t$ with $t = \mathrm{thread}(\mathrm{ld}) = \mathrm{thread}(\mathrm{st})$.
Hence, case (1) applies and gives hs-through. □

Theorem (Locality, Bouajjani, M., Mohlmann, ICALP'11):
In a minimal violation, only a single thread reorders its actions.

Proof:

Consider a minimal violation $\tau \in C_{TSO}(P)$.

Towards a contradiction,

suppose <u>at least two</u> threads delay stores.

- By lemma from last lecture,
  each store is delayed past a load of the same thread.
  Let st_2 be the overall last store
  that was delayed in τ.
  Let st_2 be from thread t_2.
  Let ld_2 be the overall last load
  that was overtaken by st_2.
  Similarly, let st_1 be the overall last store
  delayed in a thread $t_1 \neq t_2$.
  Let ld_1 be the last load overtaken by st_1.

- There are the following three situations:

(1) $\tau = \tau_1 . ld_1 . \tau_2 . st_1 . \tau_3 ld_2 . \tau_4 st_2 \; \tau_5 \in C_{TSO}(P)$

(2) $\tau = \tau_1 . ld_2 . \tau_2 \; ld_1 \; \tau_3 \; st_1 \; \tau_4 \; st_2 \; \tau_5 \in C_{TSO}(P)$

(3) $\tau = \tau_1 . ld_1 \; \tau_2 \; ld_2 \; \tau_3 \; st_1 \; \tau_4 \; st_2 \; \tau_5 \in C_{TSO}(P)$.

<u>Case (1):</u>

We argue that in this case τ is not minimal
and therefore the case does not apply.
Indeed, consider
$\tau' := \tau_1 . ld_1 . \tau_2 . st_2 . \tau_3 \in C_{TSO}(P)$.

Here, $\bar{\tau}_{st}$ contains stores of thread t_2 that were issued before st_1.

- Then:

$$\#(\bar{\tau}') < \#(\bar{\tau})$$ as the delay of st_2 over ld_2 is missing.

Moreover,

$Tr(\bar{\tau}')$ is cyclic.

The reason is that

$ld_1 \rightarrow^{+}_{hb} st_1$ through $\bar{\tau}_2$

continues to hold.

The only critical check here is

$c_i \rightarrow^{+}_{po} c_{i+1}$.

One can show that

- as long as $\bar{\tau}_2$ is not changed
- and the resulting computation $\bar{\tau}'$ is feasible ($\in C_{TSO}(P)$),

then

hb-through continues to hold.

Together, $\#(\bar{\tau}') < \#(\bar{\tau})$

and $Tr(\bar{\tau}')$ cyclic contradicts minimality of $\bar{\tau}$.

Case (2):

Again, the case would imply that $\bar{\tau}$ is not minimal and can therefore not occur.

- Starting from ld_2, thread t_2 does not do actions except delayed stores, until st_2 (this was a lemma last time).

Therefore, ld_2 and all program-order later actions of t_2 can be removed without affecting feasibility of the computation.

To be precise, we also have to remove $\bar{\tau}_5$, because other threads may load from stores of t_2 that are program-order later than ld_2:

$$\bar{\tau}' := \bar{\tau}_1 . \bar{\tau}_2 . ld_1 . \bar{\tau}_3 . st_1 . \bar{\tau}_4 . st_2 \in C_{TSO}(P).$$

- Then
$$\#(\bar{\tau}') < \#(\bar{\tau}).$$

Moreover,

$Tr(\bar{\tau}')$ is cyclic

due to $ld_1 \to^+_{hb} st_1$ through $\bar{\tau}_3$.

$\lightning$ minimality of $\bar{\tau}$.

Case (3):

Again we derive a contradiction to minimality of $\bar{\tau}$.

Remember

$$\bar{\tau} = \bar{\tau}_1 . ld_1 . \bar{\tau}_2 . ld_2 . \bar{\tau}_3 \, st_1 \, \bar{\tau}_4 \, st_2 \, \bar{\tau}_5 \in C_{TSO}(P).$$

- First, we delete $\bar{\tau}_5$.
Then we delete all actions from $\bar{\tau}_4$ that do not belong to thread t_2:

$$\bar{\tau}' := \bar{\tau}_1 . ld_1 . \bar{\tau}_2 \, ld_2 \, \bar{\tau}_3 \, st_1 \, (\bar{\tau}_4 \downarrow t_2) . st_2 \in C_{TSO}(P).$$

As this does not change $\bar{\tau}_2 \, ld_2 \, \bar{\tau}_3$,
we still have

$ld_1 \to^+_{hb} st_1$ through $\bar{\tau}_2 \, ld_2 \, \bar{\tau}_3$

and $st_1 \to^+_{po} ld_2$.

So

$Tr(\bar{\tau}')$ is cyclic.

- Moreover, deleting actions does not introduce reorderings.
  Therefore,
  $$\#(\bar{\tau}') \leq \#(\bar{\tau}).$$
  Together, also $\bar{\tau}'$ is a minimal violation.
- We apply dichotomy again
  and get
  $$ld_2 \rightarrow^+_{hb} st_2 \quad \text{through } \bar{\tau}_3.st_2.(\tau_4 \downarrow t_2).$$
  Moreover,
  $$st_2 \rightarrow^+_{po} ld_2.$$
- Now, ld_2 is the program-order last action
  of thread t_2 in $\bar{\tau}'$.
  We delete it and get
  $$\bar{\tau}'' := \bar{\tau}_1\, \bar{\iota}_2\, ld_2\, \bar{\tau}_3\, st_2 (\bar{\tau}_4 \downarrow t_2)\, .\, st_2 \in G_{TIO}(P).$$
  Then
  $$\#(\bar{\tau}'') < \#(\bar{\tau}') \leq \#(\bar{\tau}).$$
  Moreover,
  $Tr(\bar{\tau}'')$ is cyclic as the cycle
  $$st_2 \rightarrow^+_{po} ld_2 \rightarrow^+_{hb} st_2 \quad \text{remains.}$$
  $\lightning$ minimality of $\bar{\tau}$. □

10.3 Attacks on Robustness

Know: If robustness breaks,
then one thread delays actions.

Show: There are two instructions (a store and a load)
where a delay yields a cycle.

10.3 Attacks on Robustness

Goal: Rephrase robustness in terms of a simple problem:
absence of feasible attacks.

Definition (Attacks):

- An attack is a triple $A = (t_A, stinst, ldinst)$
  consisting of
  - ↳ a thread $t_A \in \overline{TID}$, called attacker,
  - ↳ a store instruction $stinst$ in t_A, and
  - ↳ a load instruction $ldinst$ in t_A.
- A TSO witness of A is a computation of the form

so that
(W1) Only the attacker delays stores
(W2) Store st_A is an instance of $stinst$.
It is the first store of the attacker
that is delayed.
Load ld_A is an instance of $ldinst$.
It is the last action of the attacker
overtaken by st_A.
  - ↳ So τ_2 contains loads, assignments, asserts, and issues,
    but no fences and stores of the attacker.
  - ↳ It may contain arbitrary actions from the other threads,
    called helpers.

(W3) For all actions act in $ld_A\ \tau_3\ st_A$ we have $ld \rightarrow_{hb}^{*} act$.
Here, an isu + st of a helper is counted as one action act.

(W4) Sequence τ_4 only consists of stores of the attacker that were issued before ld_Π and have been delayed

(W5) All these stores st satisfy
$addr(st) \neq addr(ld_\Pi)$,
which means ld_Π has not read its value early.

If a TSO witness for Π exists, the attack is called *feasible*.

Example (Dekker = 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';

There is an attack $\Pi = (t_1, st_{inst}, ld_{inst})$
with
st_{inst} = the store at l_0
ld_{inst} = the load at l_1.
A TSO witness of the attack is

$$\tau = \underbrace{(t_1, isu)}_{isu_{st_\Pi}} . \underbrace{(t_1, ld, y, 0)}_{ld_\Pi} . \underbrace{(t_2, isu) . (t_2, st, y, 1) . (t_2, ld, x, 0)}_{\tau_3} . \underbrace{(t_1, st, x, 1)}_{st_\Pi}$$

So attack Π is feasible.
The program contains a symmetric attack Π'
with t_2 as the attacker.

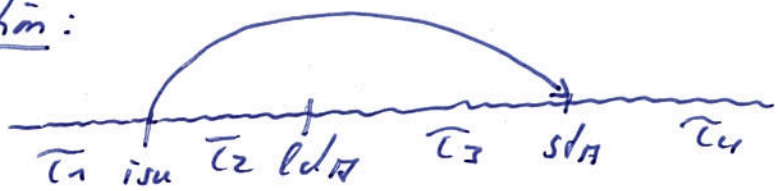
Theorem (Characterisation of robustness with attacks):
Program P is robust iff no attack is feasible.

Proof:
$\Rightarrow$: Π TSO witness comes with a happens-before cycle
$st_\Pi \rightarrow_{po}^+ ld_\Pi \rightarrow_{hb}^+ st_\Pi$.

$\Leftarrow$ Show that if P is not robust,
then there is a feasible attack.

- Among the violating computations,
we select $\bar{\tau} \in C_{TSO}(P)$ with $\#(\bar{\tau})$ minimal.
By locality, only one thread t_A uses its buffer.
Hence, (W1) holds.
- Initially, the attacker t_A executes under SC,
so stores immediately follow their issues.
↳ This computation is embedded into $\bar{\tau}_1$.
Eventually, the attacker starts delaying stores.
↳ Let st_A be the first store that is delayed.
↳ It gets delayed past several loads,
the last being ld_A. Together, we get (W2).

Illustration:



- For the helpers, note that $ld_A \to^+_{hb} st_A$ through $\bar{\tau}_3$
by dichotomy.
Consider the actions act with
$ld_A \to^*_{hb}$ act through the intermediary computation.
Then act can be moved before ld_A using dichotomy.
Hence, (W3)
- With cycle $st_A \to^*_{po} ld_A \to^+_{hb} st_A$,
$\bar{\tau}_4$ only needs to contain stores of the attacker
that have been delayed past ld_A.

These stores are non-blocking, so the helpers can stop with the last action of τ_3.
We can moreover assume ld_{17} to be the program-order last action of the attacker.
(WG) holds.

- The early read could be avoided by moving ld_{17} to τ_4.
  This would save a delay and hence contradict minimality.

Together, we have a TSO witness of the attack

$(d_{17}, st_{inst}, ld_{inst})$

where st_{inst} is the instruction of st_{17} and
ld_{inst} is the instruction of ld_{17}. □

Note:
- The number of attacks is quadratic in the size of the program.
- Enumerate them and check each for a TSO witness.
- How to? Instrumentation – next.

10.4 Instrumentation

Goal: Consider program P with attacker $\pi = (t_\pi, \text{stmt}, \text{buf})$.
Characterise TSO witnesses of π in P
by SC computations in a program P_π
that is instrumented for attacker π.

By instrumentation we mean we replace
every thread in P by a modified version.

Tricky: • Unbounded store buffers
• Unbounded happens-before dependencies.

Instrumenting the attacker:

Idea: • Emulate store buffering under SC
using auxiliary addresses
↳ When the attacker executes the delayed store st_π,
under SC it is done right behind the issue action.
↳ To mimic store buffering, st_π now accesses
an auxiliary address that helpers do not load.
↳ Indeed, in $\tilde{C}_4$ the helpers are no longer active
and hence do not access the delayed stores.
• How many auxiliary addresses?
↳ One per address in the program (last store).

Technically: • Starting from st_π to address a,
stores are replaced by st_π^{aux} to addresses (a,d). // d = delay.
• As long as address a has not been written,
(a,d) holds the initial value 0.
When the attacker stores v to address a,
we set $mem[(a,d)] = (v,d)$.
Hence, (a,d) always holds the most recent store
to address a.

- A load $r \leftarrow mem[a]$ of the attacker reads value v from the buffer whenever $mem[(a,d)] = (v,d)$. Otherwise $mem[(a,d)] = 0$, and the load obtains $v = mem[a]$ from memory.

Definition (instrumentation of the attacker):
Consider thread t_A regs r^* init lo begin $\langle inst \rangle^*$ end.
Let $A = (t_A, stinst, ldinst)$ be the attack.
The instrumentation of t_A for attack A
is the thread:

$[\![t_A]\!] :=$ thread t_A regs r^* init lo

begin
$\langle linst \rangle^*$ // Source code
$[\![stinst]\!]_{A_1}$ // Move to copy of source code
$[\![ldinst]\!]_{A_1}$
$[\![\langle linst \rangle]\!]^*_{A_2}$ // Copy of source code.

end.

with
$[\![\underbrace{l_1 : mem[e_1] \leftarrow e_2 \text{ goto } l_2}_{stinst}]\!]_{A_1} :=$ $l_1 : mem[(e_1,d)] \leftarrow (e_2,d)$ goto $\tilde{l}_x$;
$\tilde{l}_x : mem[as_A] \leftarrow e_1$ goto $\tilde{l}_2$;

$[\![\underbrace{l_1 : r \leftarrow mem[e] \text{ goto } l_2}_{ldinst}]\!]_{A_2} :=$ $\tilde{l}_1 :$ assert $mem[(e,d)] = 0$ goto $\tilde{l}_x$
$\tilde{l}_{x1} :$ $mem[bb] \leftarrow$ true goto $\tilde{l}_{x2}$;
$\tilde{l}_{x2} :$ $mem[(e,bb)] \leftarrow$ lda goto $\tilde{l}_{x3}$;

$[\![l_1 : mem[e_1] \leftarrow e_2 \text{ goto } l_2]\!]_{A_2} :=$ $\tilde{l}_1 :$ $mem[(e_1,d)] \leftarrow (e_2,d)$ goto $\tilde{l}_2$;

$[\![l_1 : r \leftarrow mem[e] \text{ goto } l_2]\!]_{A_2} :=$ $\tilde{l}_1 :$ assert $mem[(e,d)] = 0$ goto $\tilde{l}_{x1}$;
$\tilde{l}_{x1} :$ $r \leftarrow mem[e]$ goto $\tilde{l}_2$;
$\tilde{l}_1 :$ assert $mem[(e,d)] \neq 0$ goto $\tilde{l}_x$
$\tilde{l}_{x2} :$ $(r,d) \leftarrow mem[(e,d)]$; goto $\tilde{l}_2$;

$[\![l_1 : \text{local goto } l_2]\!]_{\pi_2} := \widehat{\tau_1} : \text{local goto } \widehat{l_2};$

$[\![l_1 : \text{nfence goto } l_2]\!]_{\pi_2} := /$

Comment:

- Note that the instrumentation $[\![\text{st}_i \text{ inst}]\!]_{\pi_1}$ keeps the address used in the store in a fresh address a_{st_π}.
- The instrumentation deletes fences as they forbid to delay st_π over ld_π.
- The instrumentation $[\![\text{ld}_i \text{ inst}]\!]_{\pi_1}$ checks the value is not read early. Moreover, if such a happens-before address (a, hb) to access level load, lda. It also sets a flag hb to forbid helper actions that do not contribute to happens-before path $\widehat{\tau_3}$.

Instrumenting helpers:

Idea: · How to decide whether a new action act is in happens-before relation with an earlier action act' so that $\text{ld}_\pi \xrightarrow{*}_{hb} act' \xrightarrow{*}_{hb} act$?

Need to know two facts:

↳ Has the thread of act already contributed an action act' to $\widehat{\tau_3}$?
In this case, $act' \xrightarrow{*}_{po} act$.
The information about such a contribution can be kept in the control-flow of the helper.

↳ Does $\widehat{\tau_3}$ contain a load or store access to addr(act)?

- If there was a load $act' = \text{ld}$, we can add a store $act = \text{st}$ and get $\text{ld} \xrightarrow{*}_{hb} \text{st}$
- If there was a store, we are free to add a load or a store.

↳ Need one auxiliary address (a, hb)
per address a in the program.
The addresses (a, hb) range over the domain

$$\{0, lda, sta\}$$

of access types.
It is sufficient to store the maximal access type
wrt. the ordering:

$$0 \text{ (no access)} < lda \text{ (load access)} < sta \text{ (store access)}.$$

Technically: The augmentation on a thread's contribution to $\tilde{\tau}_3$
+ access types is based on the following lemma.

Lemma:
Consider $\tilde{\tau} = \tilde{\tau}_1.\, act_1.\, \tilde{\tau}_2 \in C_{SC}(P)$
where for all $act_2 \in \tilde{\tau}_2$ we have $act_1 \rightarrow_{hb}^* act_2$.
Then $\tilde{\tau}.act$ satisfies $act_1 \rightarrow_{hb}^* act$

if
(1) $\exists\, act_2 \in act_1.\, \tilde{\tau}_2 :\ thread(act_2) = thread(act)$,
(2) act is a load whose address is stored in $act_1.\, \tilde{\tau}_2$, or
(3) act is a store (with issue) whose address
is loaded or stored in $act_1.\, \tilde{\tau}_2$.

W.R.t. this, the instrumentation of helpers is as follows.

Definition (Instrumentation of helpers):
Consider thread t regs r^* init to begin $\langle instr \rangle^*$ end.
The instrumentation of t is
$[\![t]\!]$:= thread $\tilde{t}$ regs $\tilde{r}, r^*$ init to

begin
$[\![\langle instr \rangle]\!]^*_{H_0}$ $[\![\langle ldst.instr \rangle]\!]^*_{H_1}$ $[\![\langle instr \rangle]\!]^*_{H_2}$ $[\![\langle \iota \rangle]\!]^*_{H_3}$
end

- Here, $\langle ldst inst \rangle^*$ is the subsequence of all load and store instructions. Their instrumentation $[\![\langle ldst inst \rangle]\!]^*_{H_2}$ is used to move to the code copy $[\![\langle linst \rangle]\!]^*_{H_2}$
- Let $\langle l \rangle^*$ be all labels used by the thread. The instructions $[\![\langle l \rangle]\!]^*_{H_3}$ raise a success flag when a TSO witness has been found.
- The instrumentation $[\![\langle linst \rangle]\!]^*_{H_0}$ of the original source code forces the helper to either enter the code copy $[\![\langle linst \rangle]\!]^*_{H_2}$ or stop when the hb-flag is raised.

The functions are as follows:

$[\![l_1 : instr \ \underline{goto} \ l_2]\!]_{H_0} :=$ $l_1 :$ assert $mem[hb] = 0$ goto l_x;
$l_x :$ instr goto l_2;

$[\![l_1 : r \leftarrow mem[e] \ \underline{goto} \ l_2]\!]_{H_1} :=$ $l_1 :$ assert $mem[(e, hb)] = sta$ goto $\widetilde{l_x}$;
$\widetilde{l_x} : r \leftarrow mem[e]$ goto $\widetilde{l_2}$;

$[\![l_1 : mem[e_1] \leftarrow e_2 \ \underline{goto} \ l_2]\!]_{H_1} :=$ $l_1 :$ assert $mem[(e_1, hb)] \geq lda$ goto $\widetilde{l_{x_1}}$
$\widetilde{l_{x_1}} :$ $mem[e_1] \leftarrow e_2$ goto $\widetilde{l_{x_2}}$;
$\widetilde{l_{x_2}} :$ $mem[(e_1, hb)] \leftarrow sta$ goto $\widetilde{l_2}$;

$[\![l_1 : local / \underline{mfence} \ goto \ l_2]\!]_{H_2} :=$ $\widetilde{l_1} :$ local / mfence goto $\widetilde{l_2}$;
$[\![l_1 : mem[e_1] \leftarrow e_2 \ goto \ l_2]\!]_{H_2} :=$ $\widetilde{l_1} :$ $mem[e_1] \leftarrow e_2$ goto $\widetilde{l_x}$;
$\widetilde{l_x} :$ $mem[(e_1, hb)] \leftarrow sta$ goto $\widetilde{l_2}$;

$[\![l_1 : r \leftarrow mem[e] \ goto \ l_2]\!]_{H_2} :=$ $\widetilde{l_1} :$ $\widetilde{r} \leftarrow e$ goto $\widetilde{l_{x_1}}$;
$\widetilde{l_{x_1}} :$ $r \leftarrow mem[\widetilde{r}]$ goto $\widetilde{l_{x_2}}$
$\widetilde{l_{x_2}} :$ $mem[(\widetilde{r}, hb)] \leftarrow$
$\max\{lda, mem[(\widetilde{r}, hb)]\}$
goto $\widetilde{l_2}$;

$[\![l]\!]_{H_3} ::= \tilde{l}: \tilde{r} \leftarrow mem[as|_H]$ goto $\tilde{l}_{x_1}$;

$\tilde{l}_{x_1}: \tilde{r} \leftarrow mem[(\tilde{r}, hb)]$ goto $\tilde{l}_{x_2}$;

$\tilde{l}_{x_2}:$ assert $\tilde{r} \neq 0$ goto $\tilde{l}_{x_3}$;

$\tilde{l}_{x_3}: mem[suc] \leftarrow true$ goto $\tilde{l}_{x_4}$;

Note:
In the instrumentation of loads, $[\![l_1: r \leftarrow mem[e]$ goto $l_2]\!]_{H_2}$, auxiliary register $\tilde{r}$ ensures that we do not overwrite the address given by e when modifying r (may be used within e).

Theorem (Soundness and completeness of instrumentation):
$\forall$ Hack H is feasible in program P
iff P_H reaches a goal configuration under SC.

$\forall$ goal configuration is a pair (pc, val)
with val(suc) = true.

Theorem:
- Program P is robust iff no instrumentation P_H reaches a goal configuration under SC
- If the data domain is finite and given as input, robustness is PSPACE-complete.

Proof:
Upper bound: We show that the complement of robustness, the non-robustness problem (given a program P, check that P is not robust) can be solved in non-deterministic polynomial space (NPSPACE).
By Savitch's theorem NPSPACE = PSPACE,
and hence non-robustness $\in$ PSPACE
We negate the answer and get robustness $\in$ PSPACE.

Essentially, we use that

$$\text{co-NPSPACE} = \text{co-PSPACE} = \text{PSPACE} \ (= \text{NPSPACE}).$$

To solve non-robustness in NPSPACE,
we guess a suitable attack A
and compute the linear-size instrumentation P_A.
Then we guess a reaching path in P_A.
For the path, we only need to store

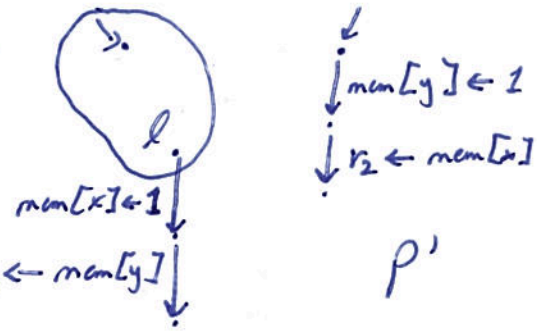
- the current configuration (works in linear space)
- the number of steps taken (works in linear space as well).

Return yes, if a goal configuration is reached.
Return no, if the search for a path deadlocks
or the number of steps exceeds the number of configurations.
For the latter, note that
there are 2^n configurations with n bits.
We need another n-bit to count to 2^n.

Lower bound: We first give a reduction of SC-reachability
in Boolean programs to non-robustness.

- Consider a single-threaded Boolean program P
with control-location ℓ

- Using a second thread and fresh variables,
we append a Dekker cycle to ℓ:



- Then ℓ is SC-reachable in P
iff P' is not robust.
- Since control-state reachability
is PSPACE-hard, so is non-robustness.

To see that also robustness is PSPACE-hard,
consider a problem Prob $\in$ PSPACE
that we want to reduce to robustness.
Since PSPACE is closed under complement,
we have

$$\text{co-Prob} \in \text{PSPACE}.$$

We just showed that non-robustness is PSPACE-hard.
Hence there is a reduction

$$f: \text{co-Prob} \to \text{non-Rob}$$

so that

instance $i \in$ co-Prob iff $f(i)$ is not robust.

Since $i \in$ co-Prob iff $i \notin$ Prob,
we have

$i \in$ Prob iff $f(i)$ is robust.

So function f is also a reduction of Prob to robustness. □