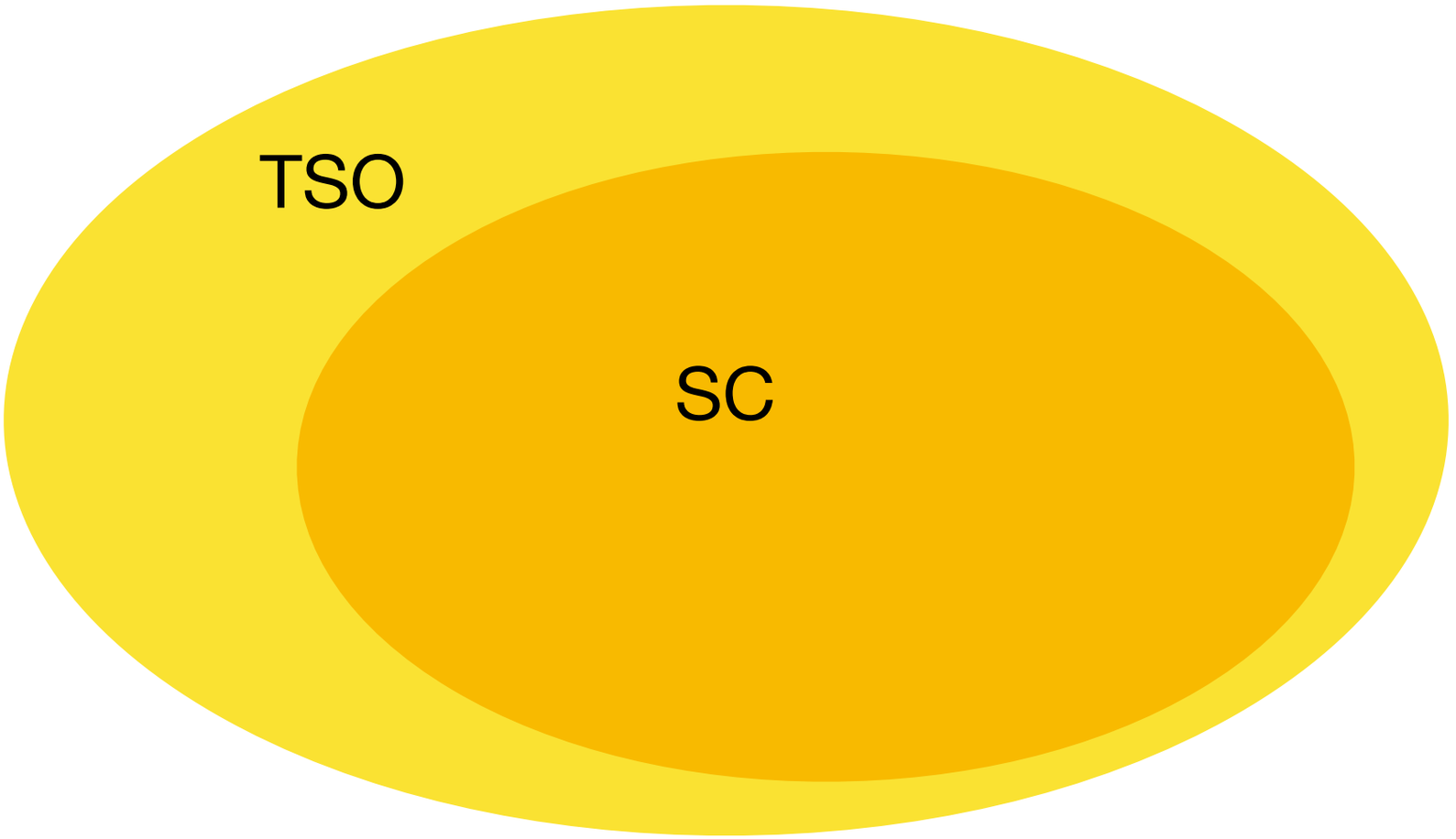
Robustness

Robustness

- We assume the programmer didn’t want relaxed behaviors
- Verification problem:
  - Specification: SC version of the program
  - Implementation: TSO version of the program
- The question of robustness can be asked for any two memory models (PSO -> SC), (PSO -> TSO), etc.
- The only practical techniques are for the simpler ones: TSO, PSO

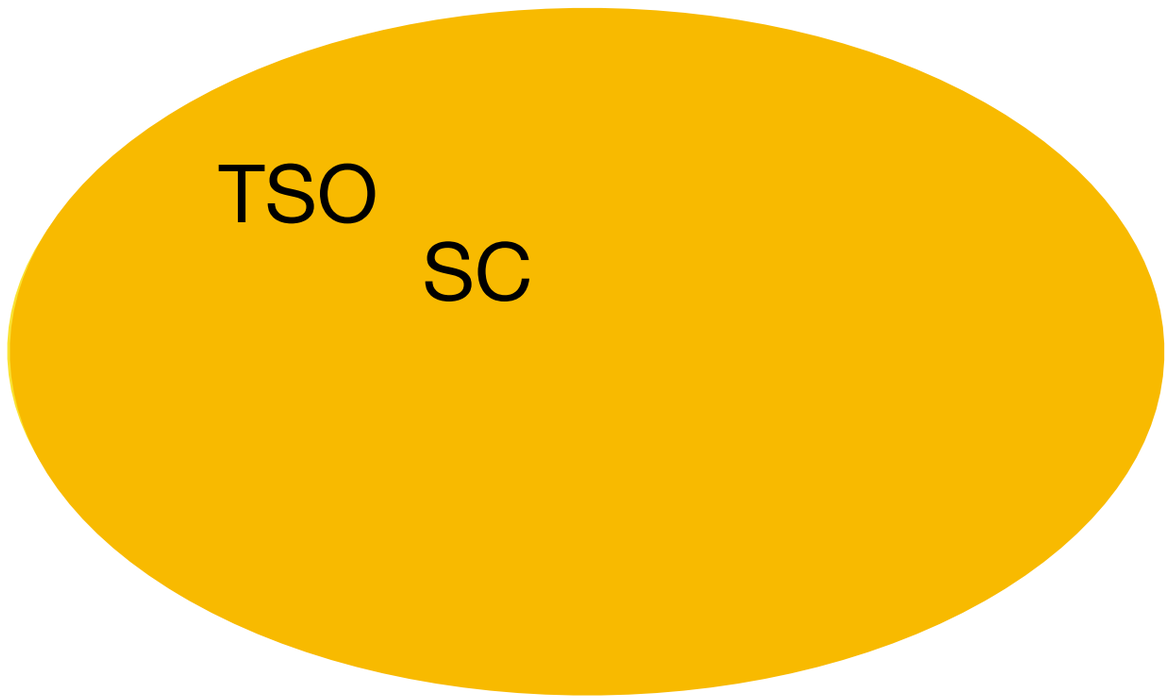
TSO Robustness

A program P is robust if all of its TSO behaviors are coincide with its SC behaviors

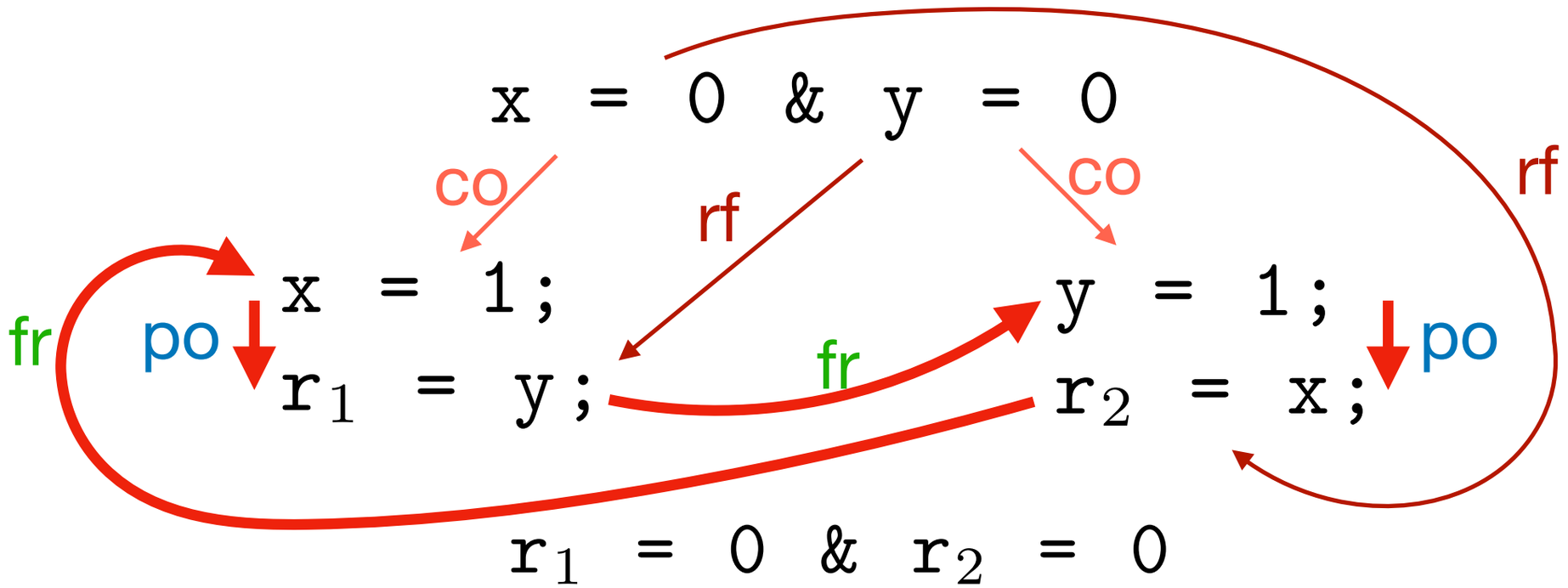


TSO Robustness

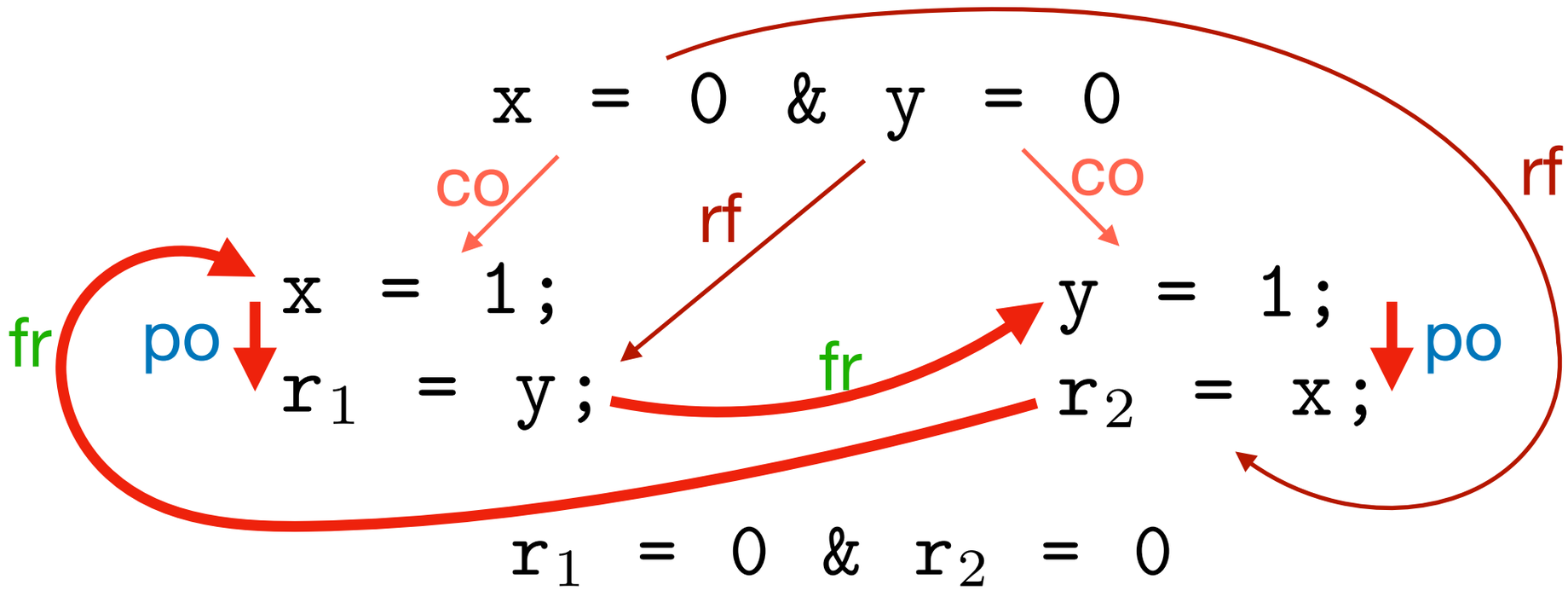
A program P is robust if all of its TSO behaviors are coincide with its SC behaviors



Shasha & Snir 1988



Shasha & Snir 1988



P is robust if all its traces contain no happens-before cycles

Checking TSO Robustness

Bouajanni, Derevenetc, Meyer 2013

- Characterize minimal SC violations under TSO
- Any program containing non-SC computations contains minimal SC violations
- Instrument the program to detect minimal SC-violations (only)
- Model check the program under SC

Minimal SC-violations

Minimal SC-violations

- A TSO execution that is non-SC

Minimal SC-violations

- A TSO execution that is non-SC
- We use the operational model with write buffers

$$\big(S, T \parallel (B, \mathbf{E}[(\lambda x\ ev)])\big) \quad \xrightarrow{\beta} \quad \big(S, T \parallel (B, \mathbf{E}[\{x/v\}e])\big)$$

$$\big(S, T \parallel (B, \mathbf{E}[(\mathsf{ref}\, v)])\big) \xrightarrow{\nu_{p,v}} \big(S \cup \{p \mapsto v\}, T \parallel (B, \mathbf{E}[p])\big) \quad p \notin \mathsf{dom}(S)$$

$$\big(S, T \parallel (B, \mathbf{E}[(p := v)])\big) \xrightarrow{\mathsf{wr}_{p,v}} \big(S, T \parallel (B \triangleleft [p \mapsto v], \mathbf{E}[()])\big)$$

$$\big(S, T \parallel (B, \mathbf{E}[(!\, p)])\big) \xrightarrow{\mathsf{rd}_{p,v}} \big(S, T \parallel (B, \mathbf{E}[v])\big) \quad B(p) = \epsilon\ \&\ S(p) = v$$

$$\big(S, T \parallel (B, \mathbf{E}[(!\, p)])\big) \xrightarrow{\mathsf{rd}_{p,v}} \big(S, T \parallel (B, \mathbf{E}[v])\big) \quad B(p) = ls :: v$$

$$\big(S, T \parallel (B, \mathbf{E}[\langle \mathtt{wr} | \mathtt{rd} \rangle])\big) \xrightarrow{\mathtt{wr}} \big(S, T \parallel (B, \mathbf{E}[()])\big) \quad \forall p, B(p) = \epsilon$$

$$\big(S, T \parallel ([p \mapsto v] \triangleright B, e)\big) \xrightarrow{\mathsf{bu}_{p,v}} \big(S[p \mapsto v], T \parallel (B, e)\big) \quad TSO$$

Minimal SC-violations

- A TSO execution that is non-SC
- We use the operational model with write buffers

$$\begin{array}{rcll}
(S, T \parallel (B, \mathbf{E}[(\lambda x\ ev)])) & \xrightarrow{\beta} & (S, T \parallel (B, \mathbf{E}[\{x/v\}e])) & \\
(S, T \parallel (B, \mathbf{E}[(\mathsf{ref}\, v)])) & \xrightarrow{\nu_{p,v}} & (S \cup \{p \mapsto v\}, T \parallel (B, \mathbf{E}[p])) & p \notin \mathsf{dom}(S) \\
(S, T \parallel (B, \mathbf{E}[(p := v)])) & \xrightarrow{\mathsf{wr}_{p,v}} & (S, T \parallel (B \triangleleft [p \mapsto v], \mathbf{E}[()])) & \\
(S, T \parallel (B, \mathbf{E}[(!\, p)])) & \xrightarrow{\mathsf{rd}_{p,v}} & (S, T \parallel (B, \mathbf{E}[v])) & B(p) = \epsilon\ \&\ S(p) = v \\
(S, T \parallel (B, \mathbf{E}[(!\, p)])) & \xrightarrow{\mathsf{rd}_{p,v}} & (S, T \parallel (B, \mathbf{E}[v])) & B(p) = ls :: v \\
(S, T \parallel (B, \mathbf{E}[\langle \mathtt{wr} | \mathtt{rd} \rangle])) & \xrightarrow{\mathtt{wr}} & (S, T \parallel (B, \mathbf{E}[()])) & \forall p, B(p) = \epsilon \\
 & & & \\
(S, T \parallel ([p \mapsto v] \triangleright B, e)) & \xrightarrow{\mathsf{bu}_{p,v}} & (S[p \mapsto v], T \parallel (B, e)) & \mathit{TSO}
\end{array}$$

Minimal SC-violations

- A TSO execution that is non-SC
- We use the operational model with write buffers
- We consider only the labels

$$(t_0, wr_{x,1}) \cdot (t_0, bu_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, bu_{y,1}) \cdot (t_1, rd_{x,1})$$

Minimal SC-violations

- A TSO execution that is non-SC
- We use the operational model with write buffers
- We consider only the labels

$$(t_0, wr_{x,1}) \cdot (t_0, bu_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, bu_{y,1}) \cdot (t_1, rd_{x,1})$$

No SC-violation

Minimal SC-violations

- A TSO execution that is non-SC
- We use the operational model with write buffers
- We consider only the labels
- It contains the minimum number of “delays” possible

Minimal SC-violations

- A TSO execution that is non-SC
- We use the operational model with write buffers
- We consider only the labels
- It contains the minimum number of “delays” possible

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1}) \cdot (t_1, bu_{y,1})$$

Minimal SC-violations

- A TSO execution that is non-SC
- We use the operational model with write buffers
- We consider only the labels
- It contains the minimum number of “delays” possible

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1}) \cdot (t_1, bu_{y,1})$$

SC-violation, but not minimal

Minimal SC-violations

- A TSO execution that is non-SC
- We use the operational model with write buffers
- We consider only the labels
- It contains the minimum number of “delays” possible

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1}) \cdot (t_1, bu_{y,1})$$

SC-violation, but not minimal

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, bu_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1})$$

Minimal SC-violations

- A TSO execution that is non-SC
- We use the operational model with write buffers
- We consider only the labels
- It contains the minimum number of “delays” possible

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1}) \cdot (t_1, bu_{y,1})$$

SC-violation, but not minimal

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, bu_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1})$$

SC-violation, minimal

Minimal SC-violations

- A TSO execution that is non-SC
- We use the operational model with write buffers
- We consider only the labels
- It contains the minimum number of “delays” possible

Theorem: In every minimal violation there is at most one thread that delays stores

Taxonomy of a TSO violation

We will call the delaying thread the attacker

$$\sigma = \sigma_1 \cdot (t_A, wr_{x,v}) \cdot \sigma_2 \cdot (t_A, rd_{y,v'}) \cdot \sigma_3 \cdot (t_A, bu_{x,v}) \cdot \sigma_4$$

Taxonomy of a TSO violation

We will call the delaying thread the attacker

SC subtrace

$$\sigma = \sigma_1 \cdot (t_A, wr_{x,v}) \cdot \sigma_2 \cdot (t_A, rd_{y,v'}) \cdot \sigma_3 \cdot (t_A, bu_{x,v}) \cdot \sigma_4$$

Taxonomy of a TSO violation

We will call the delaying thread the attacker

SC subtrace

$$\sigma = \sigma_1 \cdot (t_A, wr_{x,v}) \cdot \sigma_2 \cdot (t_A, rd_{y,v'}) \cdot \sigma_3 \cdot (t_A, bu_{x,v}) \cdot \sigma_4$$

First write of the attacker to be delayed

Taxonomy of a TSO violation

We will call the delaying thread the attacker

SC subtrace

$$\sigma = \sigma_1 \cdot (t_A, wr_{x,v}) \cdot \sigma_2 \cdot (t_A, rd_{y,v'}) \cdot \sigma_3 \cdot (t_A, bu_{x,v}) \cdot \sigma_4$$

First write of the attacker to be delayed

Last read of the attacker before updating the delayed write

Taxonomy of a TSO violation

We will call the delaying thread the attacker

SC subtrace

subtrace with no attacker actions

$$\sigma = \sigma_1 \cdot (t_A, wr_{x,v}) \cdot \sigma_2 \cdot (t_A, rd_{y,v'}) \cdot \sigma_3 \cdot (t_A, bu_{x,v}) \cdot \sigma_4$$

First write of the attacker to be delayed

Last read of the attacker before updating the delayed write

Taxonomy of a TSO violation

We will call the delaying thread the attacker

SC subtrace

subtrace with no attacker actions

$$\sigma = \sigma_1 \cdot (t_A, wr_{x,v}) \cdot \sigma_2 \cdot (t_A, rd_{y,v'}) \cdot \sigma_3 \cdot (t_A, bu_{x,v}) \cdot \sigma_4$$

First write of the attacker to be delayed

Last read of the attacker before updating the delayed write

update of the delayed write

Taxonomy of a TSO violation

We will call the delaying thread the attacker

SC subtrace

subtrace with no attacker actions

other updates of attacker

$$\sigma = \sigma_1 \cdot (t_A, wr_{x,v}) \cdot \sigma_2 \cdot (t_A, rd_{y,v'}) \cdot \sigma_3 \cdot (t_A, bu_{x,v}) \cdot \sigma_4$$

First write of the attacker to be delayed

update of the delayed write

Last read of the attacker before updating the delayed write

In Dekker

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, bu_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1})$$

In Dekker

First delayed write

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, bu_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1})$$

In Dekker

First delayed write

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, bu_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1})$$

Last read of t0 before buffer update

In Dekker

First delayed write

subtrace with no attacker actions

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, bu_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1})$$

Last read of t0 before buffer update

In Dekker

First delayed write

subtrace with no attacker actions

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, bu_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1})$$

Last read of t0 before buffer update

update of the delayed write

In Dekker

First delayed write

subtrace with no attacker actions

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, bu_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1})$$

Last read of t0 before buffer update

update of the delayed write

Now we just need to check if a TSO program can be attacked

In Dekker

First delayed write

subtrace with no attacker actions

$$(t_0, wr_{x,1}) \cdot (t_0, rd_{y,0}) \cdot (t_1, wr_{y,1}) \cdot (t_1, bu_{y,1}) \cdot (t_1, rd_{x,0}) \cdot (t_0, bu_{x,1})$$

Last read of t0 before buffer update

update of the delayed write

Now we just need to check if a TSO program can be attacked

We will call the tuple $A = (t_A, rd, bu)$ an attack, and check if it can happen

Instrumentation

1. We instrument the program
   - We pick an attack $A = (t_A, rd, bu)$
   - At the point of the write $(t_A, wr\_,\_)$ involved in the attack we instrument the program to not write in the memory but record it's own writes
   - We check that the read $(t_A, rd\_,\_)$ is reachable before the action $(t_A, bu\_,\_)$ happens
2. We use *an SC model checker* to validate the feasibility of the attack in the instrumented program
3. Query all the possible attacks!

Effectively reducing TSO Robustness to an SC reachability query!

Complexity

- finite data domain
- finite memory
- finite threads

} PSpace-Complete

Enforcing Robustness

- Compute all possible attacks
- For each feasible attack compute the set of fences that prevent it
- Find optimal set of fences using Integer Linear Programming (ILP)

Program Logics for Relaxed Memory

A Premier on Hoare-Logics

- Also known as (Floyd-)Hoare Logic
- We need a language of assertions describing property of the state at different program points
- Program variables can be used in assertions to relate the program term and the state
- Invariants: Assertions that are true of all the states in a piece of code (e.g. loop invariant, global invariant, etc.)

Assertions

$$
\begin{array}{rcl}
P & ::= & True \\
 & | & \neg P \\
 & | & P \wedge Q \\
 & | & P \vee Q \\
 & | & P \Rightarrow Q \\
 & | & x = v | \ldots \\
 & | & \exists X, P \\
 & | & \forall X, P
\end{array}
$$

Any possible state

All states that satisfy P and Q

Logical variables relate values in one state

$\{\exists X, x = X \wedge y = X + 1\}$

Substitutions

$$
\begin{array}{rcl}
\{\exists X, x = X \wedge y = X + 1\}[x \leftarrow 8] & = & \{\exists X, 8 = X \wedge y = X + 1\} \\
 & \equiv & \{y = 9\}
\end{array}
$$

Notation: $P[x \leftarrow 8] \approx [x/8]P$

The meaning of Triples

For any initial state satisfying P

The final state must satisfy Q

$$\{P\}\ c\ \{Q\} \iff \forall \sigma \vDash P,\ \text{when } (\sigma, c) \xrightarrow{*} \sigma',\ \text{we have } \sigma' \vDash Q$$

If we execute c reaching a final state

Triples can be composed to prove complex programs

$$\{Pre\}\ c_0;\ \{P\}\ c;\ \{Q\}\ c_1;\ \{Post\}$$

Check these examples

$\{x = 3\}\ x := x + 1\ \{x \leq 0\}$

$\{x = 3\}\ x := x + 1\ \{x \geq 0\}$

$\{x = 3\}\ x := x + 1; y := x\ \{y \geq 0\}$

$\{\exists X, x = X \wedge y = X + 1\}\ x := x + 1; y := x\ \{x = y\}$

$\{x = 0\}$ if $true$ then $x := x + 1$ else $x := x - 1$ fi $\{x = 1\}$

$\{x = 0\}$ if $x = 5$ then $x := x + 1$ else $x := x - 1$ fi $\{x = 1\}$

$\{x = 0\}$ while $true$ do $x := x + 1$ od $\{false\}$

Hoare Logic Rules

Hoare Logic Rules

$$\{P\}\ skip\ \{P\}$$

Hoare Logic Rules

$$\{P\}\ skip\ \{P\}$$

$$\{P[x \leftarrow e]\}\ x := e\ \{P\}$$

Hoare Logic Rules

$$\{P\}\ skip\ \{P\} \qquad \{P[x \leftarrow e]\}\ x := e\ \{P\}$$

$$\frac{\{P\}\ c_0\ \{R\} \qquad \{R\}\ c_1\ \{Q\}}{\{P\}\ c_0; c_1\ \{Q\}}$$

Hoare Logic Rules

$$\{P\}\ skip\ \{P\} \qquad \{P[x \leftarrow e]\}\ x := e\ \{P\}$$

$$\frac{\{P\}\ c_0\ \{R\} \qquad \{R\}\ c_1\ \{Q\}}{\{P\}\ c_0; c_1\ \{Q\}}$$

$$\frac{\{P \wedge b\}\ c_0\ \{Q_0\} \quad \{P \wedge \neg b\}\ c_1\ \{Q_1\}}{\{P\}\ \text{if } b \text{ then } c_0 \text{ else } c_1 \text{ fi } \{b \Rightarrow Q_0 \wedge \neg b \Rightarrow Q_1\}}$$

Hoare Logic Rules

$$\{P\}\ skip\ \{P\}$$

$$\{P[x \leftarrow e]\}\ x := e\ \{P\}$$

$$\frac{\{P\}\ c_0\ \{R\} \qquad \{R\}\ c_1\ \{Q\}}{\{P\}\ c_0; c_1\ \{Q\}}$$

$$\frac{\{P \wedge b\}\ c_0\ \{Q_0\} \quad \{P \wedge \neg b\}\ c_1\ \{Q_1\}}{\{P\}\ \text{if } b \text{ then } c_0 \text{ else } c_1 \text{ fi } \{b \Rightarrow Q_0 \wedge \neg b \Rightarrow Q_1\}}$$

$$\frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ \text{while } b \text{ do } c \text{ od } \{\neg b \wedge P\}}$$

Hoare Logic Rules

$$\{P\}\ skip\ \{P\}$$

$$\{P[x \leftarrow e]\}\ x := e\ \{P\}$$

$$\frac{\{P\}\ c_0\ \{R\} \qquad \{R\}\ c_1\ \{Q\}}{\{P\}\ c_0; c_1\ \{Q\}}$$

$$\frac{\{P \wedge b\}\ c_0\ \{Q_0\} \quad \{P \wedge \neg b\}\ c_1\ \{Q_1\}}{\{P\}\ \text{if } b \text{ then } c_0 \text{ else } c_1 \text{ fi } \{b \Rightarrow Q_0 \wedge \neg b \Rightarrow Q_1\}}$$

$$\frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ \text{while } b \text{ do } c \text{ od } \{\neg b \wedge P\}}$$

$$\frac{\{P\}\ c\ \{Q\} \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{\{P'\}\ c\ \{Q'\}}$$

A simple Hoare Logic Proof

$$\{\mathtt{x} = X \wedge \mathtt{y} = Y\}$$
$$\mathtt{aux} = \mathtt{x};$$
$$\mathtt{x} = \mathtt{y};$$

$$\mathtt{y} = \mathtt{aux};$$
$$\{\mathtt{x} = Y \wedge \mathtt{y} = X\}$$

A simple Hoare Logic Proof

$$\{\mathtt{x} = X \wedge \mathtt{y} = Y\}$$
$$\mathtt{aux} = \mathtt{x};$$
$$\{\mathtt{x} = X \wedge \mathtt{y} = Y \ \wedge \mathtt{aux} = X\}$$
$$\mathtt{x} = \mathtt{y};$$

$$\mathtt{y} = \mathtt{aux};$$
$$\{\mathtt{x} = Y \wedge \mathtt{y} = X\}$$

A simple Hoare Logic Proof

$$\{\mathtt{x} = X \wedge \mathtt{y} = Y\}$$
$$\mathtt{aux} = \mathtt{x};$$
$$\{\mathtt{x} = X \wedge \mathtt{y} = Y \ \wedge \mathtt{aux} = X\}$$
$$\mathtt{x} = \mathtt{y};$$
$$\{\mathtt{x} = Y \wedge \mathtt{y} = Y \ \wedge \mathtt{aux} = X\}$$
$$\mathtt{y} = \mathtt{aux};$$
$$\{\mathtt{x} = Y \wedge \mathtt{y} = X\}$$

The Owicky/Gries Method

- Hoare logics is insufficient to prove concurrent programs
- In 1976 Owicky and Gries provide a method to verify concurrent programs (using the cobegin/coend parallel construct)
- First method to be able to prove properties of multiprgrams

Quick intro to OG (R/G)

$$\frac{\{P_0\}\ C_0\ \{Q_0\} \quad \{P_1\}\ C_1\ \{Q_1\} \quad \text{non-inteference}}{\{P_0 \wedge P_1\}\ C_0 \| C_1\ \{Q_0 \wedge Q_1\}}$$

Quick intro to OG (R/G)

$$\frac{\{P_0\}\ C_0\ \{Q_0\} \quad \{P_1\}\ C_1\ \{Q_1\} \quad \text{non-inteference}}{\{P_0 \wedge P_1\}\ C_0 \| C_1\ \{Q_0 \wedge Q_1\}}$$

Non-Interference

- For each triple $\{p_i\}\ c_i\ \{q_i\}$ occurring in $\{P_0\}\ C_0\ \{Q_0\}$
- For each pair $\{p_j\}\ c_j$ occurring in $\{P_1\}\ C_1\ \{Q_1\}$, where is an atomic command (write, read, …)
- Show that: p_i and q_i are stable w.r.t. c_j

$$\{p_i \wedge p_j\}\ c_j\ \{p_i\}$$

Quick intro to OG (R/G)

$$\frac{\{P_0\}\ C_0\ \{Q_0\} \quad \{P_1\}\ C_1\ \{Q_1\} \quad \text{non-inteference}}{\{P_0 \wedge P_1\}\ C_0 \| C_1\ \{Q_0 \wedge Q_1\}}$$

Non-Interference

- For each triple $\{p_i\}\ c_i\ \{q_i\}$ occurring in $\{P_0\}\ C_0\ \{Q_0\}$
- For each pair $\{p_j\}\ c_j$ occurring in $\{P_1\}\ C_1\ \{Q_1\}$, where is an atomic command (write, read, …)
- Show that: p_i and q_i are stable w.r.t. c_j

$$\{p_i \wedge p_j\}\ c_j\ \{p_i\}$$

Exponential proof obligations to number of threads

To the Board!

$$\{\texttt{x} = 0 \wedge \texttt{flag} = false\}$$

$$\begin{array}{l} \texttt{x} = 100 \\ \texttt{flag} = true \end{array} \quad \| \quad \begin{array}{l} \texttt{if(flag)} \\ \quad \texttt{x} = x - 50 \end{array}$$

$$\{\texttt{x} \geq 50 \wedge \texttt{flag} = true\}$$

To the Board!

$$\{\texttt{x} = 0 \wedge \texttt{flag} = false\}$$

$$\begin{array}{l} \texttt{x} = 100 \\ \texttt{flag} = true \end{array} \quad \| \quad \begin{array}{l} \texttt{if(flag)} \\ \quad \texttt{x} = x - 50 \end{array}$$

$$\{\texttt{x} \geq 50 \wedge \texttt{flag} = true\}$$

Atomic $\longrightarrow$

$$\begin{array}{c} \{\texttt{x} = 0\} \\ \texttt{x} = \texttt{x} + 1 \quad \| \quad \texttt{x} = \texttt{x} + 1 \\ \{\texttt{x} = 2\} \end{array}$$

To the Board!

$$\{\texttt{x} = 0 \wedge \texttt{flag} = false\}$$

$$\begin{array}{l} \texttt{x} = 100 \\ \texttt{flag} = true \end{array} \quad \| \quad \begin{array}{l} \texttt{if(flag)} \\ \quad \texttt{x} = x - 50 \end{array}$$

$$\{\texttt{x} \geq 50 \wedge \texttt{flag} = true\}$$

Atomic $\longrightarrow$ $\begin{array}{c} \{\texttt{x} = 0\} \\ \texttt{x} = \texttt{x} + 1 \quad \| \quad \texttt{x} = \texttt{x} + 1 \\ \{\texttt{x} = 2\} \end{array}$

Ghost Variables

OGRA

Owicki-Gries Reasoning for Weak Memory Models
Lahav, Vafeiadis'15

OG is unsound for TSO

Lets prove that Dekker cannot happen in SC

Borrowed Owicki-Gries Reasoning for Weak Memory Models
Lahav, Vafeiadis'15

OG is unsound for TSO

Lets prove that Dekker cannot happen in SC

$$
\begin{array}{c}
\{a \neq 0\} \\
\hline
\begin{array}{l|l}
\{a \neq 0\} & \{\top\} \\
x := 1; & y := 1; \\
\{x \neq 0\} & \{y \neq 0\} \\
a := y & b := x \\
\{x \neq 0\} & \{y \neq 0 \wedge (a \neq 0 \vee b = x)\}
\end{array} \\
\hline
\{a \neq 0 \vee b \neq 0\}
\end{array}
$$

Borrowed Owicki-Gries Reasoning for Weak Memory Models
Lahav, Vafeiadis'15

Owicky/Gries for Release Acquire

- Release/Acquire is a memory model loosely based on the release/acquire fragment of C++11
- Relelase/Acquire includes TSO behaviors
- For all practical purposes we will think of TSO

OG à la Rely/Guarantee

$$R; G \vDash \{P\}\ c\ \{Q\}$$

<u>Guarantees</u>

Set of pairs: $\{P\}\ c$

- This thread affects the state by executing c when P is true

<u>Relies</u>

Set of pairs of the form: $R \nearrow Q$

- Other threads should not invalidate R
- while assuming values for variables which do not contradict Q

Stability Check

$R \nearrow Q$ is stable w.r.t. $\{P\}\ x := v$ if

$$R \wedge P \vdash [v/x]R$$

Stability Check

$R \nearrow Q$ is stable w.r.t. $\{P\}\ x := v$ if

$$R \wedge P \vdash [v/x]R$$

$R \nearrow Q$ is stable w.r.t. $\{P\}\ x := y$ if

$$R \wedge P \vdash [v_y/x]R$$

for all v_y with $Q \wedge P \nvdash y \neq v_y$

The Logic

$$\text{(SKIP)}\ \frac{\{P \nearrow P\} \leq \mathcal{R}}{\mathcal{R}; \emptyset \Vdash \{P\}\ \texttt{skip}\ \{P\}} \qquad \text{(CONSEQ)}\ \frac{\begin{array}{c}\mathcal{R}; \mathcal{G} \Vdash \{P\}\, c\, \{Q\} \\ P' \vdash P \quad Q \vdash Q' \quad \mathcal{R} \leq \mathcal{R}' \quad \mathcal{G} \leq \mathcal{G}'\end{array}}{\mathcal{R}'; \mathcal{G}' \Vdash \{P'\}\, c\, \{Q'\}}$$

$$\text{(SEQ)}\ \frac{\begin{array}{c}\mathcal{R}_1; \mathcal{G}_1 \Vdash \{P\}\, c_1\, \{R\} \\ \mathcal{R}_2; \mathcal{G}_2 \Vdash \{R\}\, c_2\, \{Q\} \qquad \mathcal{R}_1^R \vdash \mathcal{R}_2^C\end{array}}{\mathcal{R}_1 \cup \mathcal{R}_2; \mathcal{G}_1 \cup \mathcal{G}_2 \Vdash \{P\}\, c_1; c_2\, \{Q\}}$$

$$\text{(ASSN}_0\text{)}\ \frac{P \vdash Q[v/x] \qquad \{P \nearrow P, Q \nearrow (P \vee Q)\} \leq \mathcal{R}}{\mathcal{R}; \{\{P\}x := v\} \Vdash \{P\}\, x := v\, \{Q\}} \qquad \text{(ASSN}_1\text{)}\ \frac{P \vdash Q[e(y)/x] \qquad \{P \nearrow P, Q \nearrow (P \vee Q)\} \leq \mathcal{R}}{\mathcal{R}; \{\{P\}x := e(y)\} \Vdash \{P\}\, x := e(y)\, \{Q\}}$$

$$\text{(PAR)}\ \frac{\begin{array}{c}\mathcal{R}_1; \mathcal{G}_1 \Vdash \{P_1\}\, c_1\, \{Q_1\} \qquad \mathcal{R}_2; \mathcal{G}_2 \Vdash \{P_2\}\, c_2\, \{Q_2\} \\ Q_1 \wedge Q_2 \vdash Q \qquad \mathcal{R}_1; \mathcal{G}_1 \text{ and } \mathcal{R}_2; \mathcal{G}_2 \text{ are non-interfering}\end{array}}{\mathcal{R}_1 \cup \mathcal{R}_2 \cup \{Q \nearrow (\mathcal{R}_1^R \vee \mathcal{R}_2^R \vee Q)\}; \mathcal{G}_1 \cup \mathcal{G}_2 \Vdash \{P_1 \wedge P_2\}\, c_1 \parallel c_2\, \{Q\}}$$

Examples

$$
\begin{array}{l}
\{x = 0\} \\
\{\top\} \quad \| \quad \{x \neq 0 \to m = 42\} \\
m := 42; \quad \| \quad \texttt{while } x = 0 \texttt{ do skip;} \\
\{m = 42\} \quad \| \quad \{m = 42\} \\
x := 1 \quad \| \quad a := m \\
\{\top\} \quad \| \quad \{a = 42\} \\
\{a = 42\}
\end{array}
$$

$$
\begin{array}{l}
\{x = a = c = 0\} \\
\left\{\begin{array}{l}(x \neq 1 \wedge a \neq 1) \\ \quad \nearrow x \neq 1\end{array}\right\} \quad \| \quad \left\{\begin{array}{l}(x \neq 2 \wedge c \neq 2) \\ \quad \nearrow x \neq 2\end{array}\right\} \quad \| \quad \{\top\} \quad \| \quad \{\top\} \\
x := 1 \quad \| \quad x := 2 \quad \| \quad a := x; \quad \| \quad c := x; \\
\{\top\} \quad \| \quad \{\top\} \quad \| \quad \{\top\} \quad \| \quad \{\top\} \\
\quad \| \quad \quad \| \quad b := x \quad \| \quad d := x \\
\quad \| \quad \quad \| \quad \{a = 1 \wedge b = 2 \to x = 2\} \quad \| \quad \{c = 2 \wedge d = 1 \to x = 1\} \\
\{a = 1 \wedge b = 2 \wedge c = 2 \to d \neq 1\}
\end{array}
$$

Soundness

Theorem:

If $R; G \vDash \{P\}\ c\ \{Q\}$ is derivable, then $\{P\}\ c\ \{Q\}$ is a safe Hoare triple.

Other formal tools

Other formal tools

Counter-Example Guided Fence Insertion under Weak Memory Models

Parosh Aziz Abdulla
Uppsala University
parosh@it.uu.se

Mohamed Faouzi Atig
Uppsala University
mohamed_faouzi.atig@it.uu.se

Yu-Fang Chen
Academia Sinica
yfc@iis.sinica.edu.tw

Carl Leonardsson
Uppsala University
carl.leonardsson@it.uu.se

Ahmed Rezine
Uppsala University
rezine.ahmed@it.uu.se

Abstract

We give a *sound* and *complete* procedure for fence insertion for concurrent finite-state programs running under the classical TSO memory model. This model allows "write to read" relaxation corresponding to the addition of an unbounded store buffer between each processor and the main memory. We introduce a novel machine model, called the *Single-Buffer* (SB) semantics, and show that the reachability problem for a program under TSO can be reduced to the reachability problem under SB. We present a simple and effective backward reachability analysis algorithm for the latter, and propose a counter-example guided fence insertion procedure. The procedure is augmented by a *placement constraint*, that allows the user to choose the places inside the program where fences may be inserted. For a given placement constraint, the method infers automatically all minimal sets of fences that ensure correctness of the program. We have implemented a prototype and run it successfully on all standard benchmarks, together with several challenging examples that are beyond the applicability of existing methods.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification.

General Terms Verification, Theory, Reliability.

Keywords Program verification, Relaxed memory models, Infinite-state systems.

1. Introduction

Background The *Out-of-Order Execution (OoOE)* paradigm lets the scheduling of CPU instructions be governed by the availability of their operands rather than by the order in which they are issued [39]. This leads to an efficient use of clock cycles and hence an

work under the Sequential Consistency (SC) model [26] in which the program behaves according to the classical interleaving semantics. However, this is not true any more once we consider concurrent processes that share the memory. In fact, several algorithms that are designed for the synchronization of concurrent processes, such as mutual exclusion and producer-consumer protocols, are not correct in the OoOE setting. The inadequacy of the interleaving semantics in the presence of OoOE has prompted researchers to introduce *weak (or relaxed) memory models* by allowing permutations between certain types of memory operations [2, 3, 14]. Weak memory models are used in all major CPU designs including Intel x86 [22, 38], SPARC [40], and PowerPC [21]. Since the weak memory semantics adds more behavior to a program, the program may violate its specification even if it runs correctly under the SC semantics. One way to eliminate the non-desired behaviors resulting from the use of weak memory models is to insert memory *fence* instructions in the program code. A fence instruction, executed by a process, implies that no reordering is allowed between instructions issued before and after the fence instruction.

The most common relaxation corresponds to TSO (for Total Store Ordering) that is adopted by Sun's SPARC multiprocessors [40]. TSO is the kernel of many common weak memory models and is the latest formalization of the x86-tso memory model [35, 38].

Challenge Processor vendors do not provide formal definitions for the memory models of their products, but rather informal documents describing allowed/forbidden behaviors. This is not sufficient for the development of verification tools. Therefore, a substantial research effort has recently been devoted to this issue, resulting in formal (weak) memory models (both axiomatic and operational) for several different types of processors [35, 38]. In this paper, we describe how to insert sets of fences that are sufficient

Other formal tools

Counter-Example Guided Fence Insertion under Weak Memory Models

Parosh A
Uppsal
paros

Stateless Model Checking for TSO and PSO

Parosh Abdulla, Stavros Aronis, Mohammed Faouzi Atig,
Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas

Dept. of Information Technology, Uppsala University, Sweden

Abstract

We give a *sound a*
concurrent finite-st
memory model. Th
responding to the
each processor and
chine model, called
the reachability pro
to the reachability
fective backward r
propose a counter-
procedure is augm
user to choose the
inserted. For a give
matically all minin
program. We have
on all standard ben
amples that are bey

Categories and Su
ing]: Software/Prog

General Terms V

Keywords Progra
Infinite-state syster

1. Introducti

Background The
the scheduling of C
of their operands r
[39]. This leads to

Abstract. We present a technique for efficient stateless model checking of programs that execute under the relaxed memory models TSO and PSO. The basis for our technique is a novel representation of executions under TSO and PSO, called *chronological traces*. Chronological traces induce a partial order relation on relaxed memory executions, capturing dependencies that are needed to represent the interaction via shared variables. They are optimal in the sense that they only distinguish computations that are inequivalent under the widely-used representation by Shasha and Snir. This allows an optimal dynamic partial order reduction algorithm to explore a minimal number of executions while still guaranteeing full coverage. We apply our techniques to check, under the TSO and PSO memory models, LLVM assembly produced for C/pthreads programs. Our experiments show that our technique reduces the verification effort for relaxed memory models to be almost that for the standard model of sequential consistency. In many cases, our implementation significantly outperforms other comparable tools.

1 Introduction

Verification and testing of concurrent programs is difficult, since one must consider all the different ways in which instructions of different threads can be interleaved. To make matters worse, most architectures implement *relaxed memory models*, such as TSO and PSO [31,3], which make threads interact in even more and subtler ways than by standard interleaving. For example, a processor may reorder loads and stores by the same thread if they target different addresses, or it may buffer stores in a local queue.

A successful technique for finding concurrency bugs (i.e., defects that arise only under some thread schedulings), and for verifying their absence, is *stateless model checking* (SMC) [15], also known as *systematic concurrency testing* [20,34]. Starting from a test, i.e., a way to run a program and obtain some expected result, which is terminat-

Other formal tools

Counter-Example Guided Fence Insertion under Weak Memory Models

Parosh A
Uppsal
paros

Stateless Model Checking for TSO and PSO

Parosh Al
Bengt Jons

Dept. of I

Abstract

We give a *sound a*
concurrent finite-st
memory model. Th
responding to the
each processor and
chine model, called
the reachability pro
to the reachability
fective backward r
propose a counter-
procedure is augme
user to choose the
inserted. For a give
matically all minin
program. We have
on all standard ben
amples that are bey

Categories and Su
ing]: Software/Prog

General Terms V

Keywords Progra
Infinite-state syster

1. Introducti

Background The
the scheduling of C
of their operands r
[39]. This leads to

Stateless Model Checking for POWER

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson
Dept. of Information Technology, Uppsala University, Sweden

Abstract. We pre
grams that execute
our technique is a
chronological trac
laxed memory exe
the interaction via
distinguish comput
tion by Shasha and
algorithm to explor
coverage. We appl
models, LLVM as
show that our tech
els to be almost th
cases, our impleme

1 Introduction

Verification and testing
the different ways in wh
matters worse, most arc
PSO [31,3], which make
interleaving. For examp
if they target different a

A successful techniq
der some thread schedul
ing (SMC) [15], also kn
test, i.e., a way to run a

Abstract. We present the first framework for efficient application of stateless model checking (SMC) to programs running under the relaxed memory model of POWER. The framework combines several contributions. The first contribution is that we develop a scheme for systematically deriving operational execution models from existing axiomatic ones. The scheme is such that the derived execution models are well suited for efficient SMC. We apply our scheme to the axiomatic model of POWER from [7]. Our main contribution is a technique for efficient SMC, called *Relaxed Stateless Model Checking* (RSMC), which systematically explores the possible inequivalent executions of a program. RSMC is suitable for execution models obtained using our scheme. We prove that RSMC is sound and optimal for the POWER memory model, in the sense that each complete program behavior is explored exactly once. We show the feasibility of our technique by providing an implementation for programs written in C/pthreads.

1 Introduction

Verification and testing of concurrent programs is difficult, since one must consider all the different ways in which parallel threads can interact. To make matters worse, current shared-memory multicore processors, such as Intel's x86, IBM's POWER, and ARM, [28,44,27,8], achieve higher performance by implementing *relaxed memory models* that allow threads to interact in even subtler ways than by interleaving of their instructions, as would be the case in the model of *sequential consistency* (SC) [31]. Under the relaxed memory model of POWER, loads and stores to different memory lo-

Other formal tools

Counter-Example Guided Fence Insertion under Weak Memory Models

Parosh A
Uppsal
paros

Stateless Model Checking for TSO and PSO

Parosh Al
Bengt Jons

Dept. of I

Abstract

We give a *sound a*
concurrent finite-st
memory model. Th
responding to the
each processor and
chine model, called
the reachability pro
to the reachability
fective backward r
propose a counter-
procedure is augme
user to choose the
inserted. For a give
matically all minin
program. We have
on all standard ben
amples that are bey

Categories and Su
ing]: Software/Prog

General Terms V

Keywords Progra
Infinite-state syster

1. Introducti

Background The
the scheduling of
of their operands
[39]. This leads to

Stateless Model Checking for POWER

Parosh Aziz Abdul

Dept.

Abstract. We pre
grams that execute
our technique is a
chronological trac
laxed memory exe
the interaction via
distinguish comput
tion by Shasha and
algorithm to explor
coverage. We appl
models, LLVM as
show that our techn
els to be almost th
cases, our impleme

1 Introduction

Verification and testing
the different ways in wh
matters worse, most arch
PSO [31,3], which make
interleaving. For examp
if they target different a

A successful techniq
der some thread schedul
ing (SMC) [15], also kn
test, i.e., a way to run a

Abstract. We
model checkin
POWER. The
that we develo
els from existi
models are we
model of POW
SMC, called *R*
explores the po
execution mod
optimal for the
behavior is ex
providing an ir

1 Introduction

Verification and test
all the different way
current shared-mem
and ARM, [28,44,2
ory models that allo
their instructions, as
Under the relaxed m

GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation

Aaron Turon Viktor Vafeiadis Derek Dreyer
Max Planck Institute for Software Systems (MPI-SWS)
{turon,viktor,dreyer}@mpi-sws.org

Abstract

Weak memory models formalize the inconsistent behaviors that one can expect to observe in multithreaded programs running on modern hardware. In so doing, however, they complicate the already-difficult task of reasoning about correctness of concurrent code. Worse, they render impotent the sophisticated formal methods that have been developed to tame concurrency, which almost universally assume a strong (*i.e.,* sequentially consistent) memory model.

This paper introduces **GPS**, the first program logic to provide a full-fledged suite of modern verification techniques—including ghost state, protocols, *and* separation logic—for high-level, structured reasoning about weak memory. We

RAM, threads take turns interacting with it, an
ory update performed by one thread is immedi
to all other threads. Even assuming sequential
concurrent program verification is a highly chall
lem, since one must account for the myriad inte
threads. But fortunately there has been tremend
in recent years on advanced program logics a
tion tools to help tame the complexity of interle
tion [9, 11, 13, 15, 24, 29, 30, 36, 37, 39].

Unfortunately, the assumption of sequential
is unrealistically "strong": the synchronization
implement it on modern architectures precludes
piler optimizations that reorder memory opera
thus considered by many to be too expensive in

A word on Decidability

Memory Model	Name	Reach. Problem
$\{w \to r,\ RLWE\}$	TSO	decidable [3]
$TSO \cup \{w \to w\}$	-	decidable [3]
$TSO \cup \{w \to w,\ RWF\}$	PSO	decidable [**new**]
$PSO \cup \{r \to r\}$	NSW	decidable [**new**]
$TSO \cup \{r \to r/w\}$	-	undecidable [3]
$TSO \cup \{r \to w\}$	-	undecidable [**new**]
$NSW \cup \{RRWE\}$	-	undecidable [**new**]

What's Decidable about Weak Memory Models? (Extended Version)

M. F. Atig[1], A. Bouajjani[2], S. Burckhardt[3], and M. Musuvathi[3]

[1] Uppsala University, Sweden, mohamed_faouzi.atig@it.uu.se
[2] LIAFA, Paris Diderot Univ. & CNRS, France, {atig,abou}@liafa.jussieu.fr
[3] Microsoft Research Redmond, USA, {sburckha,madanm}@microsoft.com

Abstract. We investigate the decidability of the state reachability problem in finite-state programs running under weak memory models. In [3], we have shown that this problem is decidable for TSO and its extension with the write-to-write order relaxation, but beyond these models nothing is known to be decidable. Moreover, we have shown that relaxing the program order by allowing reads or writes to overtake reads leads to undecidability.
In this paper, we refine these results by sharpening the (un)decidability frontiers on both sides. On the positive side, we introduce a new memory model NSW (for non-speculative writes) that extends TSO with the write-to-write relaxation, the read-to-read relaxation, and support for partial fences. We present a backtrack-free operational model for NSW, and prove that it does not allow causal cycles (thus barring pathological out-of-thin-air effects). On the negative side, we show that adding the read-to-write relaxation to TSO causes undecidability, and that adding non-atomic writes to NSW also causes undecidability.
Our results establish that NSW is the first known hardware-centric memory model that is relaxed enough to permit both delayed execution of writes and early execution of reads for which the reachability problem is decidable.

1 Introduction

The memory consistency model (or simply, the memory model) of a shared-memory multiprocessor is a low-level programming abstraction that defines when and in what order writes performed by one processor become visible to other processors. The simplest memory model, sequential consistency [16], requires that the operations performed by the processors should appear as if these operations are interleaved in a consistent global order. Despite its simplicity and appeal, most contemporary hardware platforms support Weak (relaxed) Memory Models for performance reasons [2, 13].

The effects of weal memory models can be counterintuitive and difficult to understand even for very small programs. Not surprisingly, relaxed memory models are an

End of day 3