Lecture 4: Verification of Weak Memory Models
Part 2: Robustness against TSO

Ahmed Bouajjani

LIAFA, University Paris Diderot – Paris 7

Joint work with Roland Meyer, Egor Derevenetc (Univ. Kaiserslautern) and Eike Möhlmann (Univ. Oldenburg)

VTSA, MPI-Saarbrücken, September 2012
Dekker’s Protocol

Synchronise access of two threads to their critical sections

Dekker’s mutual exclusion protocol

\[ t_1 : q_0 \rightarrow q_1 \rightarrow cs \quad t_2 : q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow cs \]
Dekker’s Protocol

Synchronise access of two threads to their critical sections

Dekker’s mutual exclusion protocol

- **Indicate wish to enter** Write own variable $x$ to 1

$$
t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \rightarrow cs \quad t_2 : q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow cs$$

What is the semantics of this program?

Depends on the hardware architecture!
Dekker’s Protocol

Synchronise access of two threads to their critical sections

Dekker’s mutual exclusion protocol

- **Indicate wish to enter**  Write own variable $x$ to 1
- **Check no wish from partner**  Check partner variable

\[
t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \\
q_0 \xrightarrow{(r,y,0)} q_1 \xrightarrow{cs} t_2 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{q_2} cs
\]
Dekker’s Protocol

Synchronise access of two threads to their critical sections

Dekker’s mutual exclusion protocol

- **Indicate wish to enter** Write own variable $x$ to 1
- **Check no wish from partner** Check partner variable
- **Symmetry** Second thread behaves similarly

$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \quad t_2 : q_0 \xrightarrow{(w,y,1)} q_{1f} \rightarrow q_2 \xrightarrow{(r,x,0)} cs$
Dekker’s Protocol

Synchronise access of two threads to their critical sections

Dekker’s mutual exclusion protocol

▶ **Indicate wish to enter** Write own variable $x$ to 1
▶ **Check no wish from partner** Check partner variable
▶ **Symmetry** Second thread behaves similarly

$$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs$$
$$t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs$$

▶ What is the **semantics** of this program?
Dekker’s Protocol

Synchronise access of two threads to their critical sections

Dekker’s mutual exclusion protocol

► Indicate wish to enter Write own variable $x$ to 1
► Check no wish from partner Check partner variable
► Symmetry Second thread behaves similarly

$$
\begin{align*}
&t_1: q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \\
&t_2: q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs
\end{align*}
$$

► What is the semantics of this program?
► Depends on the hardware architecture!
Sequential Consistency Semantics

**Sequential Consistency memory model [Lamport 1979]**

- Threads directly write to and read from memory
- Programmers often rely on this intuitive behaviour
Sequential Consistency Semantics

Sequential Consistency memory model [Lamport 1979]

- Take view from memory

Sequential Consistency semantics of Dekker’s protocol

\[
\begin{align*}
 t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs & \quad t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs \\
\text{Next:} & \quad t_1 \text{ writes } x \text{ to 1}
\end{align*}
\]

<table>
<thead>
<tr>
<th>M</th>
<th>x = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_1 : q_0</td>
<td>y = 0</td>
</tr>
<tr>
<td>t_2 : q_0</td>
<td></td>
</tr>
</tbody>
</table>
Sequential Consistency Semantics

Sequential Consistency memory model [Lamport 1979]

- Take view from memory

\[(w, x, 1)\]

Sequential Consistency semantics of Dekker's protocol

\[t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs\]

\[t_2 : q_0 \xrightarrow{(w,y,1)} f \xrightarrow{(r,x,0)} q_1 \rightarrow q_2 \rightarrow cs\]

Next: \(t_1\) reads 0 from \(y\)

\[t_1 : q_1\]

\[x = 1\]

\[t_2 : q_0\]

\[y = 0\]
Sequential Consistency Semantics

Sequential Consistency memory model [Lamport 1979]

- Take view from memory
  
  \[(w, x, 1).(r, y, 0)\]

Sequential Consistency semantics of Dekker’s protocol

\[
t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \quad t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs
\]

Next: \(t_2\) writes \(y\) to \(1\)

\[
\begin{array}{c}
t_1 : cs \\
m \\
x = 1 \\
y = 0 \\
t_2 : q_0
\end{array}
\]
Sequential Consistency Semantics

Sequential Consistency memory model [Lamport 1979]

- Take view from memory

\[(w, x, 1).(r, y, 0).(w, y, 1)\]

Sequential Consistency semantics of Dekker’s protocol

\[\begin{align*}
t_1 : q_0 & \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \\
t_2 : q_0 & \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs
\end{align*}\]

Next: \( t_2 \) executes fence \( f \)

Next state:

\[\begin{align*}
t_1 : cs \\
t_2 : q_1
\end{align*}\]
Sequential Consistency Semantics

Sequential Consistency memory model [Lamport 1979]

- Take view from memory

\[(w, x, 1).(r, y, 0).(w, y, 1).f\]

Sequential Consistency semantics of Dekker’s protocol

\[
t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \quad t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs
\]

Next: \( t_2 \) cannot read 0 from \( x \)

\[
\begin{align*}
  t_1 : cs & \quad M \\
  x = 1 & \quad y = 1 \\
  t_2 : q_2
\end{align*}
\]
Sequential Consistency Semantics

Sequential Consistency memory model [Lamport 1979]

- Take view from memory

\[(w, x, 1).(r, y, 0).(w, y, 1).f\]

Sequential Consistency semantics of Dekker’s protocol

\[t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs\]

\[t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs\]

\[t_1 : cs\]

\[M\]

\[x = 1\]

\[t_2 : q_2\]

\[y = 1\]

Mutual exclusion holds!
Total Store Ordering Semantics

- Buffers reduce latency of memory accesses

Total Store Ordering semantics of Dekker’s protocol

\[ t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \quad t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs \]

\[ t_1 : \]
\[ \begin{array}{c}
M \\
x = 0 \\
y = 0
\end{array} \]

\[ t_2 : \]
\[ \begin{array}{c}
M \\
x = 0 \\
y = 0
\end{array} \]
Total Store Ordering Semantics

- Buffers reduce latency of memory accesses
- **Total Store Ordering** architectures have write buffers

**Total Store Ordering semantics of Dekker’s protocol**

\[ t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \quad t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs \]

\[
\begin{array}{c|c}
 t_1 : & M \\
 & x = 0 \\
 t_2 : & y = 0 \\
\end{array}
\]
Total Store Ordering semantics of Dekker’s protocol

\[ t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \]

\[ t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs \]

Next: \( t_1 \) writes \((w,x,1)\) to its buffer

\[
\begin{array}{ccc}
| t_1 : q_0 | & | M | \\
| x = 0 | & | y = 0 |
\end{array}
\]
Total Store Ordering Semantics

Total Store Ordering semantics of Dekker’s protocol

\[ t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \quad t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs \]

Next: \( t_2 \) writes \((w,y,1)\) to its buffer

\[ t_1 : q_1 \xrightarrow{(w,x,1)} M \]
\[ M \]
\[ x = 0 \]
\[ y = 0 \]

\[ t_2 : q_0 \]
Total Store Ordering Semantics

- Reads prefetch last value written to $x$ from buffer

Total Store Ordering semantics of Dekker’s protocol

$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs$  $t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs$

Next: $t_1$ fails to read $(r, y, 0)$ from its buffer

\[ t_1 : q_1 \xrightarrow{(w, x, 1)} M \quad M \]
\[ x = 0 \]
\[ y = 0 \]

$t_2 : q_1 \xrightarrow{(w, y, 1)}$
Total Store Ordering Semantics

- Reads prefetch last value written to $x$ from buffer, if exists

$$(r, y, 0)$$

Total Store Ordering semantics of Dekker’s protocol

$$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \quad t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs$$

Next: $t_1$ reads $(r, y, 0)$ from memory

$$t_1 : q_1 \xrightarrow{(w,x,1)} M \quad x = 0$$

$$t_2 : q_1 \xrightarrow{(w,y,1)} y = 0$$
Total Store Ordering Semantics

- Reads prefetch last value written to $x$ from buffer, if exists
- Fences forbid prefetches
  $$(r, y, 0)$$

Total Store Ordering semantics of Dekker’s protocol

$$t_1: q_0 \xrightarrow{(w, x, 1)} q_1 \xrightarrow{(r, y, 0)} cs$$
$$t_2: q_0 \xrightarrow{(w, y, 1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r, x, 0)} cs$$

Next: $t_2$ cannot execute fence $f$ while buffer not empty

$$t_1: cs (w, x, 1) M$$
$$x = 0$$

$$t_2: q_1 (w, y, 1)$$
$$y = 0$$
Total Store Ordering Semantics

- Reads prefetch last value written to \( x \) from buffer, if exists
- Fences forbid prefetches

\((r, y, 0)\)

Total Store Ordering semantics of Dekker’s protocol

\[
t_1: q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \quad t_2: q_0 \xrightarrow{(w,y,1)} q_1 \rightarrow q_2 \xrightarrow{(r,x,0)} cs
\]

Next: memory updates \((w, y, 1)\) from buffer of \( t_2 \)

\[
\begin{array}{c|c|c}
 t_1: cs & (w, x, 1) & M \\
 & & x = 0 \\
 t_2: q_1 & (w, y, 1) & y = 0 \\
\end{array}
\]
Total Store Ordering Semantics

- Reads prefetch last value written to \( x \) from buffer, if exists
- Fences forbid prefetches

\[(r, y, 0) \cdot (w, y, 1)\]

Total Store Ordering semantics of Dekker's protocol

\[t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs\]
\[t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs\]

Next: \( t_2 \) executes fence \( f \)

\[t_1 : cs\]
\[(w, x, 1)\]
\[M\]
\[x = 0\]

\[t_2 : q_1\]
\[y = 1\]
Total Store Ordering Semantics

- Reads prefetch last value written to $x$ from buffer, if exists
- Fences forbid prefetches

$$(r, y, 0) \cdot (w, y, 1).f$$

Total Store Ordering semantics of Dekker’s protocol:

$$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \quad t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs$$

Next: $t_2$ reads $(r, x, 0)$ from memory

$$t_1 : cs \quad (w, x, 1) \quad M \quad x = 0$$

$$t_2 : q_2 \quad y = 1$$
Total Store Ordering Semantics

- Reads prefetch last value written to \( x \) from buffer, if exists
- Fences forbid prefetches

\[(r, y, 0). (w, y, 1). f.(r, x, 0)\]

Total Store Ordering semantics of Dekker’s protocol

\[ t_1 : q_0 \xrightarrow{(w, x, 1)} q_1 \xrightarrow{(r, y, 0)} cs \quad t_2 : q_0 \xrightarrow{(w, y, 1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r, x, 0)} cs \]

Next: memory updates \((w, x, 1)\) from buffer of \( t_1 \)

\[ t_1 : cs \quad (w, x, 1) \quad \begin{array}{c} M \\ x = 0 \end{array} \]

\[ t_2 : cs \quad \begin{array}{c} y = 1 \end{array} \]
Total Store Ordering Semantics

- Reads prefetch last value written to \( x \) from buffer, if exists
- Fences forbid prefetches

\[(r, y, 0) \cdot (w, y, 1) \cdot f \cdot (r, x, 0) \cdot (w, x, 1)\]

Total Store Ordering semantics of Dekker’s protocol

\[
t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \quad t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs
\]

\[
t_1 : cs \quad \begin{array}{c} M \\ x = 1 \end{array} \quad t_2 : cs \quad \begin{array}{c} y = 1 \end{array}
\]
Total Store Ordering Semantics

- Memory sees actions **out of program order**

\[(r, y, 0) \cdot (w, y, 1) \cdot f(r, x, 0) \cdot (w, x, 1)\]

Total Store Ordering semantics of Dekker’s protocol

\[t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \quad t_2 : q_0 \xrightarrow{(w,y,1)} q_1 \xrightarrow{f} q_2 \xrightarrow{(r,x,0)} cs\]

\[\begin{array}{c}
\hline
M \\
\hline
x = 1 \\
\hline
y = 1 \\
\hline
\end{array}\]

Mutual exclusion fails!
Robustness against TSO

[Burckhardt, Musuvathi, 2008], [Owens, 2010], [Alglave, Maranget, 2011]

- TSO semantics should not introduce new visible behaviors
Robustness against TSO

[Burckhardt, Musuvathi, 2008], [Owens, 2010], [Alglave, Maranget, 2011]

- TSO semantics should not introduce new visible behaviors
- What does it mean precisely?
Robustness against TSO

[Burckhardt, Musuvathi, 2008], [Owens, 2010], [Alglave, Maranget, 2011]

- TSO semantics should not introduce new visible behaviors
- What does it mean precisely?
- State-Robustness:
  
  \textit{TSO- and SC-reachable states are the same.}
Robustness against TSO

[Burckhardt, Musuvathi, 2008], [Owens, 2010], [Alglave, Maranget, 2011]

- TSO semantics should not introduce new visible behaviors
- What does it means precisely ?
- State-Robustness:
  \[ TSO\text{-} and \ SC\text{-}reachable \ states \ are \ the \ same. \]
- Reducible to state reachability: decidable but highly complex!

[Shasha, Snir, 88]
Robustness against TSO

[Burckhardt, Musuvathi, 2008], [Owens, 2010], [Alglave, Maranget, 2011]

- TSO semantics should not introduce new visible behaviors
- What does it mean precisely?
- State-Robustness:
  \textit{TSO- and SC-reachable states are the same.}
- Reducible to state reachability: decidable but highly complex!
- Trace-Robustness:
  \textit{Preservation of the traces [Shasha, Snir, 88]}

\[\]
Robustness against TSO

[Burckhardt, Musuvathi, 2008], [Owens, 2010], [Alglave, Maranget, 2011]

- TSO semantics should not introduce new visible behaviors
- What does it mean precisely?
- State-Robustness:
  \[ TSO \text{- and SC-reachable states are the same.} \]
  
  Reducible to state reachability: decidable but highly complex!

- Trace-Robustness:
  \[ Preservation \text{ of the traces [Shasha, Snir, 88]} \]

- Checking trace-robustness is less costly than checking state-robustness!
Traces

Given a computation $\tau$, consider:

- **Program order** $\rightarrow_{po}$: Order of actions issued by one thread.
- **Store order** $\rightarrow_{st}$: Order of writes to a same variable (by different threads).
- **Source relation** $\rightarrow_{src}$: `write` is source of `load`.
- The trace $T(\tau)$ is defined by the union of $\rightarrow_{po}$, $\rightarrow_{st}$, $\rightarrow_{src}$.
Traces

Given a computation $\tau$, consider:

- **Program order** $\rightarrow_{po}$: Order of actions issued by one thread.

- **Store order** $\rightarrow_{st}$: Order of writes to a same variable (by different threads).

- **Source relation** $\rightarrow_{src}$: Write is source of load.

- The trace $T(\tau)$ is defined by the union of $\rightarrow_{po}$, $\rightarrow_{st}$, $\rightarrow_{src}$.

- Given a memory model $M$, and program $P$, $Tr_M(P)$ is the set of all traces associated with computations of $P$ under $M$.

- **Robustness problem** against TSO: $Tr_{TSO}(P) = Tr_{SC}(P)$?
Traces

Given a computation $\tau$, consider:

- **Program order** $\rightarrow_{po}$: Order of actions issued by one thread.
- **Store order** $\rightarrow_{st}$: Order of writes to a same variable (by different threads).
- **Source relation** $\rightarrow_{src}$: write is source of load.
- **The trace** $T(\tau)$ is defined by the union of $\rightarrow_{po}$, $\rightarrow_{st}$, $\rightarrow_{src}$.
- Given a memory model $M$, and program $P$, $Tr_M(P)$ is the set of all traces associated with computations of $P$ under $M$.
- **Robustness problem** against TSO: $Tr_{TSO}(P) = Tr_{SC}(P)$?
- **Conflict relation** $\rightarrow_{cf}$: load can be altered by write.
- **Happen-Before relation** $\rightarrow_{hb}$: union of all relations above.
Traces

Given a computation $\tau$, consider:

- **Program order $\rightarrow_{po}$**: Order of actions issued by one thread.
- **Store order $\rightarrow_{st}$**: Order of writes to a same variable (by different threads).
- **Source relation $\rightarrow_{src}$**: write is source of load.
- The trace $T(\tau)$ is defined by the union of $\rightarrow_{po}$, $\rightarrow_{st}$, $\rightarrow_{src}$.
- Given a memory model $M$, and program $P$, $Tr_M(P)$ is the set of all traces associated with computations of $P$ under $M$.
- **Robustness problem against TSO**: $Tr_{TSO}(P) = Tr_{SC}(P)$?
- **Conflict relation $\rightarrow_{cf}$**: load can be altered by write.
- **Happen-Before relation $\rightarrow_{hb}$**: union of all relations above.
- **Thm [SS88]**: $T(\tau) \in Tr_{SC}(P)$ if and only if $\rightarrow_{hb}$ is acyclic.
Dekker’s protocol

\[ T(\tau) \]

\[
\begin{align*}
(w, x, 1) & \quad \rightarrow \quad (w, y, 1) \\
(r, y, 0) & \quad \rightarrow \quad f \\
& \quad \rightarrow \quad (r, x, 0)
\end{align*}
\]
Dekker’s protocol is not robust, $\tau$ is a violation
Deciding Robustness

Shasha and Snir do not give an algorithm to find cyclic traces!
Deciding Robustness

Shasha and Snir do not give an algorithm to find cyclic traces!

Contribution: An Algorithm for Checking Trace-Robustness
Deciding Robustness

Shasha and Snir do not give an algorithm to find cyclic traces!

Contribution: An Algorithm for Checking Trace-Robustness

- Reduce to SC reachability in instrumented programs
Deciding Robustness

Shasha and Snir do not give an algorithm to find cyclic traces!

Contribution: An Algorithm for Checking Trace-Robustness

- Reduce to SC reachability in instrumented programs
- Source-to-source translation with linear overhead
Deciding Robustness

Shasha and Snir do not give an algorithm to find cyclic traces!

**Contribution: An Algorithm for Checking Trace-Robustness**

- Reduce to SC reachability in *instrumented programs*
- *Source-to-source* translation with *linear* overhead
- Quadratic number of reachability queries
Deciding Robustness

Shasha and Snir do not give an algorithm to find cyclic traces!

Contribution: An Algorithm for Checking Trace-Robustness

- Reduce to SC reachability in instrumented programs
- Source-to-source translation with linear overhead
- Quadratic number of reachability queries
- Works for unbounded buffers and arbitrarily many threads
Deciding Robustness

Shasha and Snir do not give an algorithm to find cyclic traces!

**Contribution: An Algorithm for Checking Trace-Robustness**

- Reduce to SC reachability in *instrumented programs*
- *Source-to-source* translation with *linear* overhead
- * Quadratic number * of reachability queries
- Works for *unbounded buffers* and *arbitrarily many threads*
- *P/EXP-SPACE*-complete
Roadmap

- Locality of robustness — only one thread uses buffers
- Robustness iff no attacks
- Find attacks with SC(!) reachability
Roadmap

- Locality of robustness — only one thread uses buffers
- Robustness iff no attacks
- Find attacks with SC(!) reachability
Goal
Show that we can restrict ourselves to violations where only one thread reorders its actions
Minimal Violations

**TSO computations from rewriting**

**Reorder** \((w, x, 1).(r, y, 0) \overset{re}{\sim} (r, y, 0).(w, x, 1)\)

**Prefetch** \((w, x, v).(r, x, v) \overset{pf}{\sim} (w, x, v)\)
Minimal Violations

TSO computations from rewriting

Reorder \((w, x, 1). (r, y, 0) \circ_{re} (r, y, 0). (w, x, 1)\)

Prefetch \((w, x, v). (r, x, v) \circ_{pf} (w, x, v)\)

Minimal violations

Intuition: violations as close to SC as possible
Minimal Violations

TSO computations from rewriting

Reorder \((w, x, 1).(r, y, 0) \blacklozenge_{re} (r, y, 0).(w, x, 1)\)

Prefetch \((w, x, v).(r, x, v) \blacklozenge_{pf} (w, x, v)\)

Minimal violations

Intuition: violations as close to SC as possible

- \(#(\tau) = \text{number of rewritings to derive } \tau\)
Minimal Violations

TSO computations from rewriting

**Reorder** \((w, x, 1). (r, y, 0) \mathbin{\overset{re}{\curvearrowright}} (r, y, 0). (w, x, 1)\)

**Prefetch** \((w, x, v). (r, x, v) \mathbin{\overset{pf}{\curvearrowright}} (w, x, v)\)

Minimal violations

Intuition: violations as close to SC as possible

- \#(\tau) = number of rewritings to derive \(\tau\)
- violation \(\tau\) minimal if there is no violation \(\tau'\) with \(\#(\tau') < \#(\tau)\)
Minimal Violations

TSO computations from rewriting

Reorder  \((w, x, 1).(r, y, 0) \xRightarrow{\text{re}} (r, y, 0).(w, x, 1)\)

Prefetch  \((w, x, v).(r, x, v) \xRightarrow{\text{pf}} (w, x, v)\)

Minimal violations

Intuition: violations as close to SC as possible

\(\#(\tau) = \text{number of rewritings to derive } \tau\)

\(\text{violation } \tau \text{ \textbf{minimal} if there is no violation } \tau' \text{ with }\)
\(\#(\tau') < \#(\tau)\)

Minimal violations have good properties!
Lemma

Consider minimal violation \( \alpha.b.\beta.a.\gamma \) where \( b \) has overtaken \( a \)
Lemma

Consider minimal violation \(\alpha.b.\beta.a.\gamma\) where \(b\) has overtaken \(a\).

Then \(b\) and \(a\) have \(\rightarrow_{\text{hb}}\) path through \(\beta\):

Example (Computation in Dekker’s protocol is minimal)

\[(r, y, 0). (w, y, 1). f. (r, x, 0). (w, x, 1)\]
Lemma

Consider minimal violation $\alpha.b.\beta.a.\gamma$ where $b$ has overtaken $a$.

Then $b$ and $a$ have $\rightarrow_{hb}$ path through $\beta$: subword $b_1 \ldots b_k$ with

$$b_i \rightarrow_{src/st/cf} b_{i+1} \quad \text{or} \quad b_i \rightarrow^+_p b_{i+1}$$
Helpful Lemma for Minimal Violations

Lemma
Consider minimal violation $\alpha.b.\beta.a.\gamma$ where $b$ has overtaken $a$
Then $b$ and $a$ have $\rightarrow_{hb}$ path through $\beta$:

$$b_i \rightarrow_{src/st/cf} b_{i+1} \quad \text{or} \quad b_i \rightarrow_{p} b_{i+1}$$

Example (Computation in Dekker’s protocol is minimal)

$$(r, y, 0). (w, y, 1). f.(r, x, 0). (w, x, 1) \rightarrow_{hb}$$
Locality of Robustness

Theorem (Locality of Robustness)

In a minimal violation, only a single thread uses rewriting
Locality of Robustness

Theorem (Locality of Robustness)

*In a minimal violation, only a single thread uses rewriting*

**Proof sketch**

Pick last writes that are overtaken in two threads $t_i$ and $t_j$: 
Locality of Robustness

Theorem (Locality of Robustness)

In a minimal violation, only a single thread uses rewriting

Proof sketch
Pick last writes that are overtaken in two threads \( t_i \) and \( t_j \):
Case 1: no interference

\[
\begin{array}{c}
\text{———} \quad r_j \quad \text{———} \quad w_j \quad \text{———} \quad r_i \quad \text{———} \quad w_i \quad \text{———}
\end{array}
\]
Locality of Robustness

Theorem (Locality of Robustness)
In a minimal violation, only a single thread uses rewriting

Proof sketch
Pick last writes that are overtaken in two threads \( t_i \) and \( t_j \):
Case 1: no interference

Lemma: happens before cycle \( r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j \)
Theorem (Locality of Robustness)

In a minimal violation, only a single thread uses rewriting

Proof sketch
Pick last writes that are overtaken in two threads \( t_i \) and \( t_j \):

Case 1: no interference

\[
\begin{array}{cccc}
\quad & r_j & \leftarrow & w_j & \quad & r_i & \leftarrow & w_i & \quad
\end{array}
\]

Lemma: happens before cycle \( r_j \rightarrow_{hb}^+ w_j \rightarrow_p^+ r_j \)
Read \( r_i \) not involved, delete everything from \( r_i \) on
Locality of Robustness

Theorem (Locality of Robustness)

In a minimal violation, only a single thread uses rewriting

Proof sketch
Pick last writes that are overtaken in two threads $t_i$ and $t_j$:
Case 1: no interference

\[ \overbrace{\hphantom{w_j}}^{r_j} \quad \overbrace{\hphantom{w_i}}^{w_j} \quad \hphantom{w_i} \]

Lemma: happens before cycle $r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j$
Read $r_i$ not involved, delete everything from $r_i$ on
Saves a reordering, contradiction to minimality
Locality of Robustness

Theorem (Locality of Robustness)

*In a minimal violation, only a single thread uses rewriting*

Proof sketch
Pick last writes that are overtaken in two threads \( t_i \) and \( t_j \):
Case 2: overlap

\[ \begin{array}{c}
\text{\_\_\_\_\_\_\_\_ r}_i \quad \text{\_\_\_\_\_\_\_\_ r}_j \quad \text{\_\_\_\_\_\_\_\_ w}_j \quad \text{\_\_\_\_\_\_\_\_ w}_i \quad \text{\_\_\_\_\_\_\_\_} \\
\end{array} \]
Locality of Robustness

Theorem (Locality of Robustness)

In a minimal violation, only a single thread uses rewriting

Proof sketch
Pick last writes that are overtaken in two threads $t_i$ and $t_j$:
Case 2: overlap

\[ r_i \rightarrow r_j \rightarrow w_j \rightarrow w_i \]

Argumentation similar, delete again $r_i$
Locality of Robustness

Theorem (Locality of Robustness)

In a minimal violation, only a single thread uses rewriting

Proof sketch
Pick last writes that are overtaken in two threads $t_i$ and $t_j$:
Case 3: interference

\[ r_j \rightarrow r_i \rightarrow w_j \rightarrow w_i \]

Lemma: happens before cycle $r_j \rightarrow + \rightarrow h \rightarrow w_j \rightarrow + \rightarrow p \rightarrow r_i$.
Only thread $t_i$ may contribute, delete rest.

Lemma: happens before cycle $r_i \rightarrow + \rightarrow h \rightarrow w_i \rightarrow + \rightarrow p \rightarrow r_i$.
Read $r_j$ not on this cycle, delete it, contradiction.
Locality of Robustness

Theorem (Locality of Robustness)

*In a minimal violation, only a single thread uses rewriting*

Proof sketch
Pick last writes that are overtaken in two threads $t_i$ and $t_j$:

Case 3: interference

```
________ r_j <-> r_i <-> w_j <-> w_i ________
```

Lemma: happens before cycle $r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j$
Locality of Robustness

Theorem (Locality of Robustness)

In a minimal violation, only a single thread uses rewriting

Proof sketch

Pick last writes that are overtaken in two threads $t_i$ and $t_j$:

Case 3: interference

 Lemma: happens before cycle $r_j \rightarrow_{hb}^+ w_j \rightarrow_{p}^+ r_j$

Only thread $t_i$ may contribute, delete rest
Locality of Robustness

Theorem (Locality of Robustness)

*In a minimal violation, only a single thread uses rewriting*

Proof sketch

Pick last writes that are overtaken in two threads $t_i$ and $t_j$:

Case 3: interference

![Diagram showing the interference between writes]

Lemma: happens before cycle $r_j \rightarrow^{hb} w_j \rightarrow^{p} r_j$

Only thread $t_i$ may contribute, delete rest

Lemma: happens before cycle $r_i \rightarrow^{hb} w_i \rightarrow^{p} r_i$
**Locality of Robustness**

**Theorem (Locality of Robustness)**

*In a minimal violation, only a single thread uses rewriting*

**Proof sketch**

Pick last writes that are overtaken in two threads $t_i$ and $t_j$:

Case 3: interference

![Diagram](image)

Lemma: happens before cycle $r_j \rightarrow_{hb}^+ w_j \rightarrow_{p}^+ r_j$

Only thread $t_i$ may contribute, delete rest

Lemma: happens before cycle $r_i \rightarrow_{hb}^+ w_i \rightarrow_{p}^+ r_i$

Read $r_j$ not on this cycle, delete it, **contradiction**
Roadmap

- Locality of robustness — only one thread uses buffers
- Robustness iff no attacks
- Find attacks with SC(!) reachability
Characterization of Robustness via Attacks

Goal
Reformulate Robustness in terms of a simpler problem:

absence of feasible attacks
Observation

If Prog not robust, there are these violation:

\[
\alpha \quad r \quad \rho \quad r \quad \beta \quad w \quad \omega
\]

Attacker The thread that reorders reads: only 1 by locality
Characterization of Robustness via Attacks

Observation
If Prog not robust, there are these violation:

Attacker The thread that reorders reads: only 1 by locality
Helpers Remaining threads close cycle: $r \rightarrow_{hb}^{+} w \quad w \rightarrow_{p}^{+} r$
Characterization of Robustness via Attacks

Observation
If Prog not robust, there are these violation:

Attacker  The thread that reorders reads: only 1 by locality
Helpers   Remaining threads close cycle: \( r \xrightarrow{\text{+}}_{hb} w \) \( w \xrightarrow{\text{+}}_{p} r \)

Example (Violation in Dekker’s protocol)

\[(r, y, 0).(w, y, 1).f.(r, x, 0).(w, x, 1) \xrightarrow{\text{+}}_{hb} \]
Characterization of Robustness via Attacks

Observation
If Prog not robust, there are these violation:

\[ \alpha \xrightarrow{r} \rho \xrightarrow{\beta} w \]\n
Attacker
The thread that reorders reads: only 1 by locality

Helpers
Remaining threads close cycle: \( r \rightarrow_{hb}^+ w \) \( w \rightarrow_p^+ r \)

Intuition
Two data races \( r, \text{first}(\beta) \) and \( \text{last}(\beta), w \)
Characterization of Robustness via Attacks

Idea

- Fix thread, write instruction, read instruction
- Given these parameters, find a violation as above
Characterization of Robustness via Attacks

Idea

- Fix thread, write instruction, read instruction
- Given these parameters, find a violation as above

Definition (Attack)

An attack is a triple $A = (\text{thread}, \text{write}, \text{read})$.

A TSO witness for attack $A$ is a computation as above:

\[ \alpha \rightarrow r \rightarrow \rho \rightarrow r \rightarrow \beta \rightarrow w \rightarrow \omega \]
Characterization of Robustness via Attacks

Idea

- Fix thread, write instruction, read instruction
- Given these parameters, find a violation as above

Definition (Attack)

An **attack** is a triple $A = (\text{thread, write, read})$. A TSO witness for attack $A$ is a computation as above:

![Diagram of TSO witness]

Theorem (Complete Characterization of Robustness)

*Program Prog is robust if and only if no attack has a TSO witness.*
Characterization of Robustness via Attacks

Idea

- Fix thread, write instruction, read instruction
- Given these parameters, find a violation as above

Definition (Attack)

An attack is a triple $A = (\text{thread}, \text{write}, \text{read})$. A TSO witness for attack $A$ is a computation as above:

Theorem (Complete Characterization of Robustness)

Program Prog is robust if and only if no attack has a TSO witness. The number of attacks is quadratic in the size of Prog.
Roadmap

- Locality of robustness — only one thread uses buffers
- Robustness iff no attacks
- Find attacks with SC(!) reachability
Fix an attack $A = (\text{thread, write, read})$

**Goal**

TSO witnesses for $A$ considerably restrict reorderings, enough to find TSO witnesses with SC reachability
Finding TSO witnesses with SC reachability

Idea
Turn TSO witness into an SC computation:
Finding TSO witnesses with SC reachability

Idea
Turn TSO witness into an SC computation:

\[
\begin{align*}
\alpha & \quad r \quad \rho \quad r \quad \omega \\
\end{align*}
\]

Let attacker execute under SC

\[
\begin{align*}
\alpha & \quad w \cdot r \quad \rho \quad \omega \quad r \quad \beta \\
\end{align*}
\]
Finding TSO witnesses with SC reachability

Idea

Turn TSO witness into an SC computation:

Let attacker execute under SC

Problem  Writes may conflict with helper reads
Finding TSO witnesses with SC reachability

Idea
Turn TSO witness into an SC computation:

Let attacker execute under SC
Problem  Writes may conflict with helper reads
Solution  Hide them from other threads
Finding TSO witnesses with SC reachability

Instrumentation

$$\alpha \cdot w_{loc} \cdot r \cdot \omega_{loc} \cdot r \cdot \beta$$

SC computation $\in \text{Prog}_A$ that is instrumented for attack $A$
Finding TSO witnesses with SC reachability

Instrumentation

\[ \alpha \mathtt{w_{loc}} \cdot r \quad \rho \quad \mathtt{w_{loc}} \quad r \quad \beta \]

SC computation \( \in \text{Prog}_A \) that is instrumented for attack \( A \)

- **Attacker:**
  - Hide delayed writes
  - Check that reads can move:
    - no fences, reads and prefetches have correct values
    - *Only need the last written value on each variable*

- **Helpers:** check their actions form a happen-before path

- **Size of Prog}_A is linear in size of Prog.
Finding TSO witnesses with SC reachability

Instrumentation

\[
\alpha \stackrel{w_{loc}}{\rightarrow} \rho \uplus \omega_{loc} \stackrel{r}{\rightarrow} \beta
\]

SC computation $\in \text{Prog}_A$ that is instrumented for attack $A$

- **Attacker:**
  - Hide delayed writes
  - Check that reads can move:
    no fences, reads and prefetches have correct values
    *Only need the last written value on each variable*

- **Helpers:** check their actions form a happen-before path
- **Size of Prog$_A$ is linear** in size of Prog.

**Theorem (Soundness and Completeness)**

*Attack $A$ has a TSO witness iff Prog$_A$ reaches goal state under SC.*
End of Lecture 4:

- **Locality**: focus on reorderings of one thread.
- Check existence of feasible **attacks**.
- Attacks can be found with **SC reachability**, in parallel.

- Implementation using **SPIN**. (Prototype tool: **Trencher**.)
- Experiments: Mutex protocols, lock-free stack, work stealing queue, non-blocking write protocol, etc. Reachability queries are solved in few seconds.
- Can be extended to **NSW**. What about Power, ARM?
End of Lecture 4:

- **Locality**: focus on reorderings of one thread.
- Check existence of feasible attacks.
- Attacks can be found with SC reachability, in parallel.
- Trace-robustness is as complex as SC reachability.
- Holds for programs with parametric number of threads.

Implementation using SPIN. (Prototype tool: Trencher.)

Experiments: Mutex protocols, lock-free stack, work stealing queue, non-blocking write protocol, etc. Reachability queries are solved in few seconds.

Can be extended to NSW. What about Power, ARM?
End of Lecture 4:

- **Locality**: focus on reorderings of one thread.
- Check existence of feasible **attacks**.
- Attacks can be found with **SC reachability**, in parallel.
- **Trace-robustness** is as complex as SC reachability.
- Holds for programs with **parametric** number of threads.
- Can be used for **fence insertion**: Compute a set of fence locations that is irreducible, and of minimal size.
End of Lecture 4:

▶ Locality: focus on reorderings of one thread.
▶ Check existence of feasible attacks.
▶ Attacks can be found with SC reachability, in parallel.
▶ Trace-robustness is as complex as SC reachability.
▶ Holds for programs with parametric number of threads.
▶ Can be used for fence insertion: Compute a set of fence locations that is irreducible, and of minimal size.
▶ Implementation using SPIN. (Prototype tool: Trencher.)
▶ Experiments: Mutex protocols, lock-free stack, work stealing queue, non-blocking write protocol, etc. Reachability queries are solved in few seconds.
End of Lecture 4:

- **Locality**: focus on reorderings of one thread.
- Check existence of feasible **attacks**.
- Attacks can be found with **SC reachability**, in **parallel**.
- **Trace-robustness** is as complex as **SC reachability**.
- Holds for programs with **parametric** number of threads.
- Can be used for **fence insertion**: Compute a set of fence locations that is irreducible, and of minimal size.
- **Implementation** using SPIN. (Prototype tool: TRENCHER.)
- **Experiments**: Mutex protocols, lock-free stack, work stealing queue, non-blocking write protocol, etc. Reachability queries are solved in few seconds.
- Can be extended to NSW. What about Power, ARM?
The Programming Model: Assembler

\[ \langle prog \rangle ::= \text{prog} \langle \text{pid} \rangle \langle \text{thread} \rangle^* \]
\[ \langle thrd \rangle ::= \text{thread} \langle \text{tid} \rangle \text{regs} \langle \text{reg} \rangle^* \text{init} \langle \text{label} \rangle \text{begin} \langle \text{linst} \rangle^* \text{end} \]
\[ \langle linst \rangle ::= \langle \text{label} \rangle::= \langle \text{inst} \rangle; \text{goto} \langle \text{label} \rangle \]
\[ \langle inst \rangle ::= \langle \text{reg} \rangle \leftarrow \text{mem}[\langle \text{expr} \rangle] | \text{mem}[\langle \text{expr} \rangle] \leftarrow \langle \text{expr} \rangle | \text{mfence} \]
\[ | \langle \text{reg} \rangle \leftarrow \langle \text{expr} \rangle | \text{if} \langle \text{expr} \rangle \]
\[ \langle expr \rangle ::= \langle \text{fun} \rangle(\langle \text{reg} \rangle^*) \]
### Experiments

#### Spin as backend model checker

<table>
<thead>
<tr>
<th>Prog.</th>
<th>T</th>
<th>L</th>
<th>I</th>
<th>PA</th>
<th>IA1</th>
<th>IA2</th>
<th>FA</th>
<th>F</th>
<th>Spin</th>
</tr>
</thead>
<tbody>
<tr>
<td>PetNR</td>
<td>2</td>
<td>14</td>
<td>18</td>
<td>23</td>
<td>2</td>
<td>12</td>
<td>9</td>
<td>2</td>
<td>0.7</td>
</tr>
<tr>
<td>PetR</td>
<td>2</td>
<td>16</td>
<td>20</td>
<td>12</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>DekNR</td>
<td>2</td>
<td>24</td>
<td>30</td>
<td>119</td>
<td>15</td>
<td>33</td>
<td>71</td>
<td>4</td>
<td>3.5</td>
</tr>
<tr>
<td>DekR</td>
<td>2</td>
<td>32</td>
<td>38</td>
<td>30</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>LamNR</td>
<td>3</td>
<td>33</td>
<td>36</td>
<td>36</td>
<td>9</td>
<td>15</td>
<td>12</td>
<td>6</td>
<td>1.1</td>
</tr>
<tr>
<td>LamR</td>
<td>3</td>
<td>39</td>
<td>42</td>
<td>27</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>LFSR</td>
<td>4</td>
<td>46</td>
<td>50</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>CLHLock</td>
<td>7</td>
<td>62</td>
<td>58</td>
<td>54</td>
<td>48</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0.4</td>
</tr>
<tr>
<td>MCSLock</td>
<td>4</td>
<td>52</td>
<td>50</td>
<td>30</td>
<td>26</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td>NBW5</td>
<td>3</td>
<td>25</td>
<td>22</td>
<td>9</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0.1</td>
</tr>
<tr>
<td>ParNR</td>
<td>2</td>
<td>9</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>ParR</td>
<td>2</td>
<td>10</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>WSQ</td>
<td>5</td>
<td>86</td>
<td>78</td>
<td>147</td>
<td>137</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0.7</td>
</tr>
</tbody>
</table>