Verification of Concurrent Systems under Weak Consistency Models

Checking Robustness against TSO

Ahmed Bouajjani

University of Paris

MPRI, Paris, January 2020

Lecture 2 (cont’ed)
Client

Service

Infrastructure

CLM

Imp2

Imp1/Spec

ICM'

ICM

Guarantee

Assume

refines?
satisfies?

Assume

weaker

Assume

Assume

Assume

Assume

Guarantee

Guarantee

Guarantee

Guarantee
Articles


What’s Decidable about Weak Memory Models, M-F. Atig, A. Bouajjani, S. Burckhardt, M. Musuvathi, ESOP 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

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

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

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

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
- **Symmetry**  Second thread behaves similarly

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

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

- What is the **semantics** of this program?
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

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

- 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

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

Next: \( t_1 \) writes \( x \) to 1

\[
\begin{array}{c}
t_1 : q_0 \\
t_2 : q_0
\end{array}
\]

\[
\begin{array}{c}
\begin{bmatrix}
M \\
x = 0 \\
y = 0
\end{bmatrix}
\end{array}
\]
Sequential Consistency Semantics

Sequential Consistency memory model [Lamport 1979]

- Take view from memory

\[(w, x, 1)\]

Sequential Consistency semantics of Dekker’s protocol

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

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

\[
\begin{array}{|c|c|}
\hline
M & \hphantom{y=0} \\
\hline
x & 1 \\
\hline
y & 0 \\
\hline
\end{array}
\]
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

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

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

\[
\begin{array}{|c|}
\hline
M \\
\hline
x = 1 \\
\hline
y = 0 \\
\hline
\end{array}
\]

\[ t_1 : cs \]

\[ t_2 : q_0 \]
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

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

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

\[
\begin{align*}
t_1 & : cs \\
t_2 & : q_1
\end{align*}
\]
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 \\

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

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

\[
M
\]
\[
\begin{align*}
x &= 1 \\
y &= 1
\end{align*}
\]
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

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

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

\[t_1 : cs\]

\[M\]

\[x = 1\]

\[t_2 : q_2\]

\[y = 1\]

Mutual exclusion holds!
Total Store Ordering Semantics

- Buffers reduce latency of memory accesses

Total Store Ordering semantics of Dekker’s protocol

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

\[
\begin{array}{c|c}
\text{t}_1 & M \\
\hline
x &= 0 \\
\text{t}_2 & y = 0
\end{array}
\]
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

\[
\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 \\

t_1 & : \\
    M & \\
    x & = 0 \\
    t_2 & : \\
    y & = 0
\end{align*}
\]
Total Store Ordering Semantics

Total Store Ordering semantics of Dekker’s protocol

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

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

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

\begin{align*}
  t_1 : q_0 & \quad M \\
             & \quad x = 0 \\
  t_2 : q_0 & \quad y = 0
\end{align*}
Total Store Ordering Semantics

Total Store Ordering semantics of Dekker’s protocol

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

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

\[ t_1 : q_1 \]
\[ \begin{array}{c}
(w, x, 1) \\
M \\
x = 0 \\
y = 0
\end{array} \]

\[ t_2 : q_0 \]
Total Store Ordering Semantics

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

Total Store Ordering semantics of Dekker’s protocol

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

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

$t_1 : q_1 \xrightarrow{(w, x, 1)} M$
$t_2 : q_1 \xrightarrow{(w, y, 1)}$

$M_0 : x = 0$

$M_0 : y = 0$
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:

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

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

$t_1 : q_1 \xrightarrow{(w, x, 1)} M$
$x = 0$
$t_2 : q_1 \xrightarrow{(w, y, 1)} y = 0$
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

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

Next: \( t_2 \) cannot execute fence \( f \) while buffer not empty

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

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

$(r, y, 0)$

Total Store Ordering semantics of Dekker’s protocol

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

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

<table>
<thead>
<tr>
<th>$t_1 : cs$</th>
<th>$(w, x, 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>$x = 0$</td>
</tr>
<tr>
<td>$t_2 : q_1$</td>
<td>$(w, y, 1)$</td>
</tr>
</tbody>
</table>
| $y = 0$   | }
Total Store Ordering Semantics

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

$\langle r, y, 0 \rangle . \langle w, y, 1 \rangle$

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 $f$

$t_1 : cs$

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

$t_2 : q_1$

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

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

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

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

<table>
<thead>
<tr>
<th>$t_1 : cs$</th>
<th>$M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(w, x, 1)$</td>
<td>$x = 0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$t_2 : q_2$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = 1$</td>
<td></td>
</tr>
</tbody>
</table>
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

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

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

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

$t_1 : cs$

$$(w, x, 1) \quad M$$

$x = 0$

$t_2 : cs$

$$y = 1$$
Total Store Ordering Semantics

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

$$(r, y, 0). (w, y, 1). f. (r, x, 0). (w, x, 1)$$

Total Store Ordering semantics of Dekker’s protocol

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

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

$$t_1 : cs$$

$$t_2 : cs$$

$$M$$

$$x = 1$$

$$y = 1$$
Total Store Ordering Semantics

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

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

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

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

\[t_1 : cs \quad M \quad x = 1 \quad y = 1 \quad \text{Mutual exclusion fails!} \]

\[t_2 : cs \quad M \quad x = 1 \quad y = 1 \quad \text{Mutual exclusion fails!} \]
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 means precisely?
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:
  \[ TSO \text{- and } SC \text{-reachable states are the same.} \]
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:
  \textit{TSO- and SC-reachable states are the same.}
- Reducible to state reachability: decidable but highly complex!
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-reaching 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:
  \( TSO\)- 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]

- 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]:
  \[ T(\tau) \in Tr_{SC}(P) \text{ if and only if } \rightarrow_{hb} \text{ is acyclic.} \]
Example

Dekker's protocol

\[ T(\tau) \]

\[ (w, x, 1) \]

\[ (r, y, 0) \]

\[ (w, y, 1) \]

\[ f \]

\[ (r, x, 0) \]
Example

Dekker’s protocol

\[ T(\tau) \]

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

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
Minimal Violations

Goal
Show that we can restrict ourselves to violations where only one thread reorders its actions
Minimal Violations

**TSO computations from rewriting**

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

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

**TSO computations from rewriting**

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

**Prefetch** \((w, x, v). (r, x, v) \underset{pf}{\bowtie} (w, x, v)\)

**Minimal violations**

Intuition: violations as close to SC as possible
Minimal Violations

TSO computations from rewriting

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

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

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** \((w, x, 1). (r, y, 0) \equiv_{re} (r, y, 0). (w, x, 1)\)

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

**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 \((w, x, 1).(r, y, 0) \odot_{re} (r, y, 0).(w, x, 1)\)

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

Minimal violations

Intuition: violations as close to SC as possible

- \#(\tau) = \text{number of rewritings to derive } \tau
- violation \(\tau\) minimal if there is no violation \(\tau'\) 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 $\rightarrow_{hb}$ path through $\beta$: 
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 $\beta$:

$$\begin{align*}
b_i &\rightarrow_{src/st/cf} b_{i+1} \quad \text{or} \quad b_i &\rightarrow_p^+ b_{i+1}
\end{align*}$$

Example (Computation in Dekker’s protocol is minimal)

$$\underbrace{(r, y, 0).(w, y, 1).f.(r, x, 0).(w, x, 1)}_{\rightarrow_{hb}}$$
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

```
    r_j  w_j  r_i  w_i
```

Lemma: happens before cycle $r_j !+ h b w_j !+ p r_j ! + r_i ! + w_i ! +$.

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 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

\[
\begin{array}{cccc}
\text{Case 1: no interference} & r_j & \leftarrow & w_j & \rightarrow & r_i & \leftarrow & w_i
\end{array}
\]

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

\[
\text{Lemma: happens before cycle } r_j \rightarrow_{hb}^+ w_j \rightarrow_{p}^+ r_j
\]

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

```
    _____  r_i  _____  r_j  _____  w_j  _____  w_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 2: overlap

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

\[ \begin{array}{cccccccc}
& & r_j & & r_i & & w_j & & w_i \\
\end{array} \]
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 3: interference

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

\[
\text{Lemma: happens before cycle } r_j \rightarrow_{hb}^+ w_j \rightarrow_{p}^+ r_j
\]

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

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

```
\[ r_i \quad w_j \quad w_i \]
```

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 \( \text{iff} \) 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
Characterization of Robustness via Attacks

Observation
If Prog not robust, there are these violation:

\[
\alpha \xrightarrow{r} \rho \xrightarrow{r} \beta \xrightarrow{w} \omega
\]

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

Observation
If Prog not robust, there are these violation:

\[ \alpha \xrightarrow{r} \rho \xrightarrow{r} \beta \xrightarrow{w} \omega \]

**Attacker**  The thread that reorders reads: only 1 by locality

**Helpers**  Remaining threads close cycle: \( r \rightarrow_{hb} w \) \( w \rightarrow_{p} r \)
Characterization of Robustness via Attacks

Observation
If Prog not robust, there are these violations:

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

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

Example (Violation in Dekker’s protocol)

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

Observation
If Prog not robust, there are these violation:

\[ \alpha \xrightarrow{r} \rho \xrightarrow{r} \beta \xrightarrow{w} \omega \]

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

Intuition
Two data races  \( r, first(\beta) \) and \( last(\beta), w \)
Idea

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

\[
\begin{array}{ccccc}
\alpha & r & \rho & r & w \\
& & \beta & & \\
& & & & \omega
\end{array}
\]

Theorem (Complete Characterization of Robustness)

Program \( \text{Prog} \) 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:

$$
\alpha \xrightarrow{r} \rho \xrightarrow{r} \beta \xrightarrow{w} \omega
$$

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 $A = (\text{thread}, \text{write}, \text{read})$

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:

\[ r \rightarrow r \rightarrow \omega \]

Let attacker execute under SC

Problem
Writes may conflict with helper reads

Solution
Hide them from other threads
Finding TSO witnesses with SC reachability

Idea
Turn TSO witness into an SC computation:

Let attacker execute under SC
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

\[
\begin{align*}
\alpha & \quad w_{loc} \cdot r \quad \rho \quad \omega_{loc} \quad r \quad \beta \\
\end{align*}
\]

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

Instrumentation

\[
\alpha \cdot w_{loc} \cdot r \cdot \rho \sqsubseteq \omega_{loc} \cdot r \cdot \beta
\]

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 \text{ is linear} in size of Prog.
Finding TSO witnesses with SC reachability

Instrumentation

\[ \alpha \cdot W_{\text{loc}} \cdot r \cdot \rho \iff \omega_{\text{loc}} \cdot r \cdot \beta \]

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.
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

\[
\langle \text{prog}\rangle ::= \text{prog} \langle \text{pid} \rangle \langle \text{thread} \rangle^*
\]

\[
\langle \text{thr}d\rangle ::= \text{thread} \langle \text{tid} \rangle \text{regs} \langle \text{reg} \rangle^* \text{init} \langle \text{label} \rangle \text{begin} \langle \text{lin}st \rangle^* \text{end}
\]

\[
\langle \text{lin}st\rangle ::= \langle \text{label} \rangle: \langle \text{inst} \rangle; \text{goto} \langle \text{label} \rangle
\]

\[
\langle \text{inst} \rangle ::= \langle \text{reg} \rangle \leftarrow \text{mem}[^{\langle \text{expr} \rangle}] \mid \text{mem}[^{\langle \text{expr} \rangle}] \leftarrow \langle \text{expr} \rangle \mid \text{mfence} \\
\mid \langle \text{reg} \rangle \leftarrow \langle \text{expr} \rangle \mid \text{if} \langle \text{expr} \rangle
\]

\[
\langle \text{expr} \rangle ::= \langle \text{fun} \rangle(\langle \text{reg} \rangle^*)
\]
## 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>