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

$t_1 : q_0 \longrightarrow q_1 \longrightarrow cs \quad t_2 : \mathbf{q_0} \longrightarrow \mathbf{q_1} \to \mathbf{q_2} \longrightarrow \mathbf{cs}$

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

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

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

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

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

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

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

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

- 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

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

- 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

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

Next: t_1 writes x to 1

$t_1 : q_0$

$t_2 : \mathbf{q_0}$

M
$x = 0$
$y = 0$

Sequential Consistency Semantics

Sequential Consistency memory model [Lamport 1979]

- Take view from memory

$$(w, x, 1)$$

Sequential Consistency semantics of Dekker's protocol

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

Next: t_1 reads 0 from y

$t_1 : q_1$

$t_2 : \mathbf{q_0}$

$$\boxed{\begin{array}{c} M \\ x = 1 \\ y = 0 \end{array}}$$

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

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

Next: t_2 writes y to 1

$t_1 : cs$

$t_2 : \mathbf{q_0}$

$$\boxed{\begin{array}{c} M \\ x = 1 \\ y = 0 \end{array}}$$

Sequential Consistency Semantics

Sequential Consistency memory model [Lamport 1979]

- Take view from memory

$$(w,x,1).(r,y,0).(\mathbf{w},\mathbf{y},\mathbf{1})$$

Sequential Consistency semantics of Dekker's protocol

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

Next: t_2 executes fence **f**

$$t_1 : cs \qquad t_2 : \mathbf{q_1} \qquad \boxed{\begin{array}{c} M \\ x = 1 \\ y = 1 \end{array}}$$

Sequential Consistency Semantics

Sequential Consistency memory model [Lamport 1979]

- Take view from memory

$$(w, x, 1).(r, y, 0).(\mathbf{w}, \mathbf{y}, \mathbf{1}).\mathbf{f}$$

Sequential Consistency semantics of Dekker's protocol

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

Next: t_2 cannot read 0 from x

$t_1 : cs$

$t_2 : \mathbf{q_2}$

$$\boxed{\begin{array}{c} M \\ x = 1 \\ y = 1 \end{array}}$$

Sequential Consistency Semantics

Sequential Consistency memory model [Lamport 1979]

- Take view from memory

$$(w, x, 1).(r, y, 0).(\mathbf{w}, \mathbf{y}, \mathbf{1}).\mathbf{f}$$

Sequential Consistency semantics of Dekker's protocol

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

$t_1 : cs$

$t_2 : \mathbf{q_2}$

$$\boxed{\begin{array}{c} M \\ x = 1 \\ y = 1 \end{array}}$$

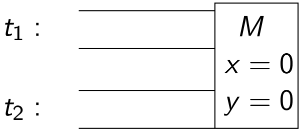
Mutual exclusion holds!

Total Store Ordering Semantics

- Buffers reduce latency of memory accesses

Total Store Ordering semantics of Dekker's protocol

$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$

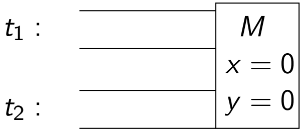


Total Store Ordering Semantics

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

Total Store Ordering semantics of Dekker's protocol

$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$

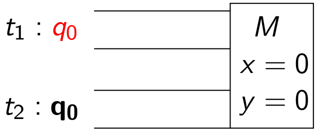


Total Store Ordering Semantics

Total Store Ordering semantics of Dekker's protocol

$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$

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

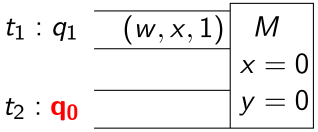


Total Store Ordering Semantics

Total Store Ordering semantics of Dekker's protocol

$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$

Next: t_2 writes $(\mathbf{w}, \mathbf{y}, \mathbf{1})$ to its buffer



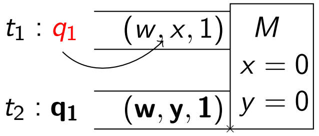
Total Store Ordering Semantics

- Reads prefetch last value written to x from buffer

Total Store Ordering semantics of Dekker's protocol

$$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$$

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



Total Store Ordering Semantics

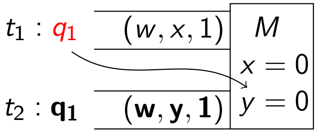
- Reads prefetch last value written to x from buffer, if exists

$(r, y, 0)$

Total Store Ordering semantics of Dekker's protocol

$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$

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



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

$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$

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

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

Total Store Ordering Semantics

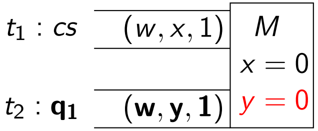
- Reads prefetch last value written to x from buffer, if exists
- Fences forbid prefetches

$(r, y, 0)$

Total Store Ordering semantics of Dekker's protocol

$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$

Next: memory updates $(\mathbf{w}, \mathbf{y}, \mathbf{1})$ from buffer of t_2



Total Store Ordering Semantics

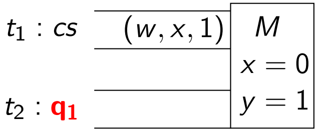
- Reads prefetch last value written to x from buffer, if exists
- Fences forbid prefetches

$$\color{red}{(r, y, 0)}\color{black}\,.(\mathbf{w}, \mathbf{y}, \mathbf{1})$$

Total Store Ordering semantics of Dekker's protocol

$$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \color{red}{\mathbf{q}_1}\color{black} \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$$

Next: t_2 executes fence $\mathbf{f}$



Total Store Ordering Semantics

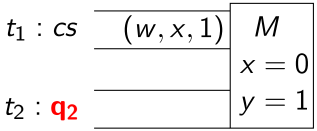
- Reads prefetch last value written to x from buffer, if exists
- Fences forbid prefetches

$$(r, y, 0)\,.(\mathbf{w}, \mathbf{y}, \mathbf{1}).\mathbf{f}$$

Total Store Ordering semantics of Dekker's protocol

$$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$$

Next: t_2 reads $(\mathbf{r}, \mathbf{x}, \mathbf{0})$ from memory



Total Store Ordering Semantics

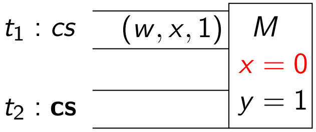
- Reads prefetch last value written to x from buffer, if exists
- Fences forbid prefetches

$$(r, y, 0)\,.(\mathbf{w}, \mathbf{y}, \mathbf{1}).\mathbf{f}.(\mathbf{r}, \mathbf{x}, \mathbf{0})$$

Total Store Ordering semantics of Dekker's protocol

$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs$ $\quad$ $t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$

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



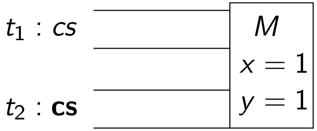
Total Store Ordering Semantics

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

$$(r, y, 0)\,.(\mathbf{w}, \mathbf{y}, \mathbf{1}).\mathbf{f}.(\mathbf{r}, \mathbf{x}, \mathbf{0})\,.(w, x, 1)$$

Total Store Ordering semantics of Dekker's protocol

$$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$$



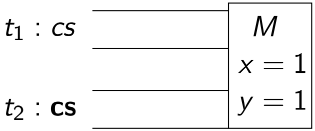
Total Store Ordering Semantics

- Memory sees actions out of program order

$$\begin{tikzcd}[column sep=small] {\color{red}(r,y,0)} \arrow[r, phantom, "."] & (\mathbf{w},\mathbf{y},\mathbf{1}).\mathbf{f}.(\mathbf{r},\mathbf{x},\mathbf{0}) \arrow[r, phantom, "."] & {\color{red}(w,x,1)} \arrow[ll, bend right, start anchor=north east, end anchor=north east] \end{tikzcd}$$

Total Store Ordering semantics of Dekker's protocol

$t_1 : q_0 \xrightarrow{(w,x,1)} q_1 \xrightarrow{(r,y,0)} cs \qquad t_2 : \mathbf{q}_0 \xrightarrow{(\mathbf{w},\mathbf{y},\mathbf{1})} \mathbf{q}_1 \xrightarrow{\mathbf{f}} \mathbf{q}_2 \xrightarrow{(\mathbf{r},\mathbf{x},\mathbf{0})} \mathbf{cs}$



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- 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- 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 means precisely ?
- State-Robustness:

  TSO- and SC-reachable states are the same.
- Reducible to state reachability: decidable but highly complex!
- Trace-Robustness:

  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 means precisely ?
- State-Robustness:
  TSO- and SC-reachable states are the same.
- Reducible to state reachability: decidable but highly complex!
- Trace-Robustness:
  Preservation of the traces [Shasha, Snir, 88]
- Checking trace-robustness is less costly than checking state-robustness!

Traces

Given a computation τ, consider:

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

Traces

Given a computation τ, consider:

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

- Program order $\rightarrow_{\texttt{po}}$: Order of actions issued by one thread.
- Store order $\rightarrow_{\texttt{st}}$: Order of writes to a same variable (by different threads).
- Source relation $\rightarrow_{\texttt{src}}$: *write* is source of *load*.
- The trace $T(\tau)$ is defined by the union of $\rightarrow_{\texttt{po}}$, $\rightarrow_{\texttt{st}}$, $\rightarrow_{\texttt{src}}$.
- 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 $\rightarrow_{\texttt{cf}}$: *load* can be altered by *write*.
- Happen-Before relation $\rightarrow_{\texttt{hb}}$: union of all relations above.

Traces

Given a computation τ, consider:

- Program order $\rightarrow_{\texttt{po}}$: Order of actions issued by one thread.
- Store order $\rightarrow_{\texttt{st}}$: Order of writes to a same variable (by different threads).
- Source relation $\rightarrow_{\texttt{src}}$: *write* is source of *load*.
- The trace $T(\tau)$ is defined by the union of $\rightarrow_{\texttt{po}}$, $\rightarrow_{\texttt{st}}$, $\rightarrow_{\texttt{src}}$.
- 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 $\rightarrow_{\texttt{cf}}$: *load* can be altered by *write*.
- Happen-Before relation $\rightarrow_{\texttt{hb}}$: union of all relations above.
- **Thm** [SS88]:

  $T(\tau) \in Tr_{SC}(P)$ *if and only if* $\rightarrow_{\texttt{hb}}$ *is acyclic.*

Example

Dekker's protocol

$$\begin{tikzcd}
T(\tau) & (w, x, 1) \arrow[d] & (\mathbf{w}, \mathbf{y}, \mathbf{1}) \arrow[d] \\
 & (r, y, 0) \arrow[ur] & \mathbf{f} \arrow[d] \\
 & & (\mathbf{r}, \mathbf{x}, \mathbf{0}) \arrow[uul]
\end{tikzcd}$$

Example

Dekker's protocol

$$\begin{tikzcd}
T(\tau) & (w,x,1) \arrow[d] & & (\mathbf{w},\mathbf{y},\mathbf{1}) \arrow[d] \\
 & (r,y,0) \arrow[rru] & & \mathbf{f} \arrow[d] \\
 & & & (\mathbf{r},\mathbf{x},\mathbf{0}) \arrow[lluu]
\end{tikzcd}$$

Dekker's protocol is not robust, τ 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 $(w,x,1).(r,y,0) \curvearrowright_{re} (r,y,0).(w,x,1)$

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

Minimal Violations

TSO computations from rewriting

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

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

Minimal violations

Intuition: violations as close to SC as possible

Minimal Violations

TSO computations from rewriting

Reorder $(w, x, 1).(r, y, 0) \curvearrowright_{re} (r, y, 0).(w, x, 1)$
Prefetch $(w, x, v).(r, x, v) \curvearrowright_{pf} (w, x, v)$

Minimal violations

Intuition: violations as close to SC as possible

- $\#(\tau)$ = number of rewritings to derive τ

Minimal Violations

TSO computations from rewriting

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

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

Minimal violations

Intuition: violations as close to SC as possible

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

Minimal Violations

TSO computations from rewriting

Reorder $(w, x, 1).(r, y, 0) \curvearrowright_{re} (r, y, 0).(w, x, 1)$
Prefetch $(w, x, v).(r, x, v) \curvearrowright_{pf} (w, x, v)$

Minimal violations

Intuition: violations as close to SC as possible

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

Minimal violations have good properties!

Helpful Lemma for Minimal Violations

Lemma

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

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* β*:*

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* β*:* *subword* $b_1 \ldots b_k$ *with*

$$b_i \rightarrow_{src/st/cf} b_{i+1} \qquad \text{or} \qquad b_i \rightarrow_p^+ b_{i+1}$$

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 β:

$$b_i \rightarrow_{src/st/cf} b_{i+1} \qquad \text{or} \qquad b_i \rightarrow_p^+ b_{i+1}$$

Example (Computation in Dekker's protocol is minimal)

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

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

$$\begin{tikzcd}
{} \arrow[r, dash] & r_j \arrow[r, dash] & w_j \arrow[l, bend right] \arrow[r, dash] & r_i \arrow[r, dash] & w_i \arrow[l, bend right] \arrow[r, dash] & {}
\end{tikzcd}$$

Locality of Robustness

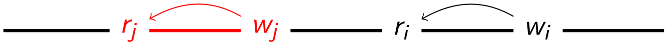
Theorem (Locality of Robustness)

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 $r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j$

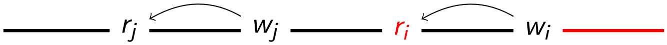
Locality of Robustness

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 $r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j$
Read r_i not involved, delete everything from r_i on

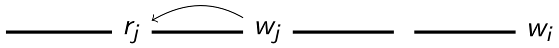
Locality of Robustness

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



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

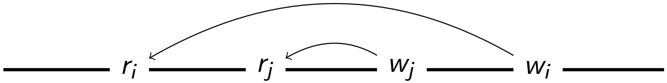
Theorem (Locality of Robustness)

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



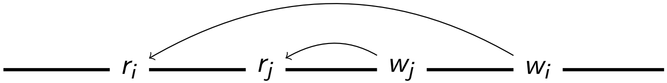
Locality of Robustness

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

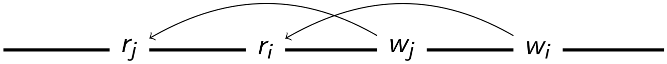
Locality of Robustness

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



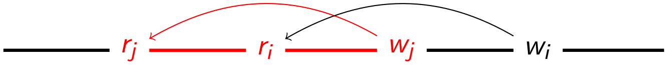
Locality of Robustness

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 $r_j \rightarrow^{+}_{hb} w_j \rightarrow^{+}_{p} r_j$

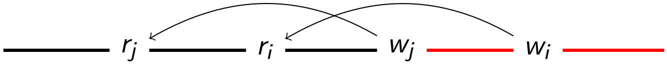
Locality of Robustness

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



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

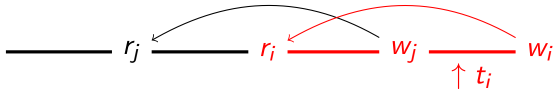
Theorem (Locality of Robustness)

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 $r_i \rightarrow_{hb}^{+} w_i \rightarrow_{p}^{+} r_i$

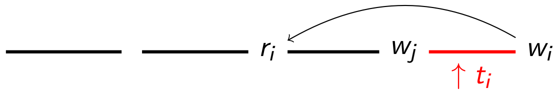
Locality of Robustness

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



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

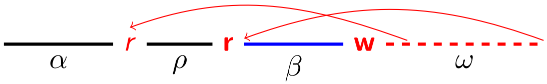
Goal

Reformulate Robustness in terms of a simpler problem:

absence of feasible attacks

Characterization of Robustness via Attacks

Observation
If Prog not robust, there are these violation:

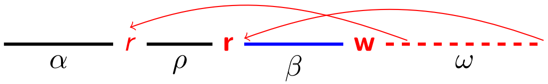


Attacker The thread that reorders reads: only 1 by locality

Characterization of Robustness via Attacks

Observation

If Prog not robust, there are these violation:



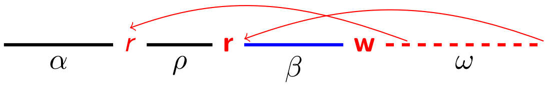
Attacker The thread that reorders reads: only 1 by locality

Helpers Remaining threads close cycle: $\mathbf{r} \rightarrow_{hb}^{+} \mathbf{w}$ $\mathbf{w} \rightarrow_{p}^{+} \mathbf{r}$

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: $\mathbf{r} \rightarrow_{hb}^{+} \mathbf{w} \ \mathbf{w} \rightarrow_{p}^{+} \mathbf{r}$

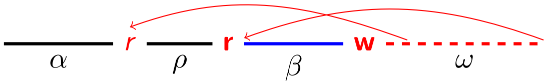
Example (Violation in Dekker's protocol)

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

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: $\mathbf{r} \rightarrow_{hb}^{+} \mathbf{w}$ $\mathbf{w} \rightarrow_{p}^{+} \mathbf{r}$

Intuition

Two data races $\mathbf{r}$, *first*(β) and *last*$(\beta), \mathbf{w}$

Characterization of Robustness via Attacks

Idea

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

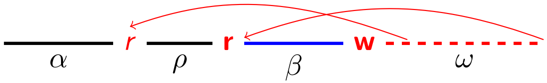
Characterization of Robustness via Attacks

Idea

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

Definition (Attack)

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



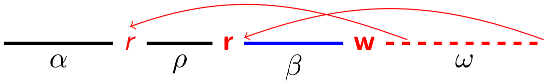
Characterization of Robustness via Attacks

Idea

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

Definition (Attack)

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

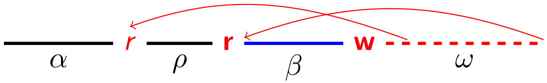
Characterization of Robustness via Attacks

Idea

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

Definition (Attack)

An attack is a triple $A = (thread, write, read)$.
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

Finding TSO witnesses with SC reachability

Fix an attack $A = (\mathit{thread}, \mathit{write}, \mathit{read})$

Goal

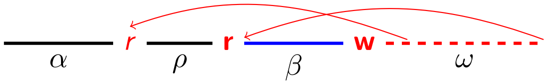
TSO witnesses for A considerably restrict reorderings,

enough to find TSO witnesses with SC reachability

Finding TSO witnesses with SC reachability

Idea

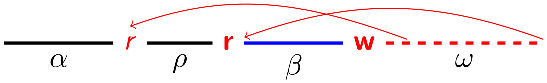
Turn TSO witness into an SC computation:



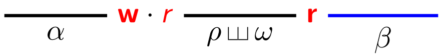
Finding TSO witnesses with SC reachability

Idea

Turn TSO witness into an SC computation:



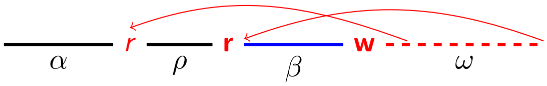
Let attacker execute under SC



Finding TSO witnesses with SC reachability

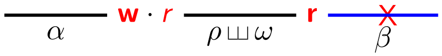
Idea

Turn TSO witness into an SC computation:



Let attacker execute under SC

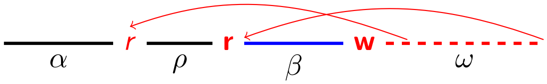
Problem Writes may conflict with helper reads



Finding TSO witnesses with SC reachability

Idea

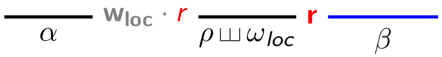
Turn TSO witness into an SC computation:



Let attacker execute under SC

Problem Writes may conflict with helper reads

Solution Hide them from other threads


α
$\mathbf{w}_{\mathbf{loc}} \cdot r$
$\rho \sqcup \omega_{loc}$
$\mathbf{r}$
β

Finding TSO witnesses with SC reachability

Instrumentation

$$\underset{\alpha}{\underline{\qquad\qquad}}\ \mathbf{w_{loc}} \cdot r\ \underset{\rho \sqcup \omega_{loc}}{\underline{\qquad\qquad}}\ \mathbf{r}\ \underset{\beta}{\underline{\qquad\qquad}}$$

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

Finding TSO witnesses with SC reachability

Instrumentation

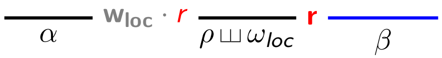
$$\underset{\alpha}{\rule{2cm}{1pt}}\ \mathbf{w_{loc}} \cdot r\ \underset{\rho \sqcup\!\sqcup\, \omega_{loc}}{\rule{2cm}{1pt}}\ \mathbf{r}\ \underset{\beta}{\rule{2cm}{1pt}}$$

SC computation $\in$ 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



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.

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

$$
\begin{array}{lcl}
\langle \mathit{prog} \rangle & ::= & \text{prog } \langle \text{pid} \rangle \ \langle \text{thread} \rangle^* \\
\langle \mathit{thrd} \rangle & ::= & \text{thread } \langle \text{tid} \rangle \text{ regs } \langle \text{reg} \rangle^* \text{ init } \langle \text{label} \rangle \text{ begin } \langle \text{linst} \rangle^* \text{ end} \\
\langle \mathit{linst} \rangle & ::= & \langle \text{label} \rangle\text{: } \langle \text{inst} \rangle\text{; goto } \langle \text{label} \rangle \\
\langle \mathit{inst} \rangle & ::= & \langle \text{reg} \rangle \leftarrow \text{mem}[\langle \text{expr} \rangle] \mid \text{mem}[\langle \text{expr} \rangle] \leftarrow \langle \text{expr} \rangle \mid \text{mfence} \\
 & & \mid \langle \text{reg} \rangle \leftarrow \langle \text{expr} \rangle \mid \text{if } \langle \text{expr} \rangle \\
\langle \mathit{expr} \rangle & ::= & \langle \text{fun} \rangle(\langle \text{reg} \rangle^*)
\end{array}
$$

Experiments

Spin as backend model checker

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