Computer-aided Verification and Construction under Relaxed Memory Models

Graduate School Weak Consistency (weacon)

Roland Meyer

Technische Universität Kaiserslautern
Concurrent Programs with Shared Memory

- Finite number of shared variables \( \{x, y, x_1, \ldots\} \)
- Finite data domain \( \{d, d_0, d_1, \ldots\} \)
- Finite number of finite-control threads \( T_1, \ldots, T_n \) with operations:
  \( w(x, d), \quad r(x, d) \)

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
</tr>
<tr>
<td>( b : if(y == 0){</td>
<td>( q : if(x == 0){</td>
</tr>
<tr>
<td>c : crit. sect. 1</td>
<td>r : crit. sect. 2</td>
</tr>
<tr>
<td>d : }</td>
<td>s : }</td>
</tr>
</tbody>
</table>

Dekker’s mutual exclusion protocol.
Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical **interleaving semantics**
  - Computations of different threads are **shuffled**
  - Program order is **preserved** for each thread

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a: x = 1)</td>
<td>(b: if(y == 0){)</td>
</tr>
<tr>
<td>(c: \text{crit. sect. 1})</td>
<td>(d: })</td>
</tr>
<tr>
<td>(p: y = 1)</td>
<td>(q: if(x == 0){)</td>
</tr>
<tr>
<td>(r: \text{crit. sect. 2})</td>
<td>(s: })</td>
</tr>
</tbody>
</table>

\[Mem\]
\[
x = y = 0
\]

\[Thread 1\]
\[
\begin{align*}
pc &= a \\
x &= 0
\end{align*}
\]

\[Thread 2\]
\[
\begin{align*}
pc &= p \\
y &= 0
\end{align*}
\]
Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical **interleaving semantics**
  - Computations of different threads are **shuffled**
  - Program order is **preserved** for each thread

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a: x = 1)</td>
<td>(p: y = 1)</td>
<td>(x)</td>
</tr>
<tr>
<td>(b: \text{if}(y == 0))</td>
<td>(q: \text{if}(x == 0))</td>
<td>(pc = b)</td>
</tr>
<tr>
<td>(c: \text{crit. sect. 1})</td>
<td>(r: \text{crit. sect. 2})</td>
<td>(y)</td>
</tr>
<tr>
<td>(d: )</td>
<td>(s: )</td>
<td>(0)</td>
</tr>
</tbody>
</table>

\(pc = p\)
Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical **interleaving semantics**
  - Computations of different threads are shuffled
  - Program order is preserved for each thread

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a : x = 1)</td>
<td>(p : y = 1)</td>
<td>(x)</td>
</tr>
<tr>
<td>(b : if(y == 0))</td>
<td>(q : if(x == 0))</td>
<td>(pc = c)</td>
</tr>
<tr>
<td>(c : crit. sect. 1)</td>
<td>(r : crit. sect. 2)</td>
<td>(y)</td>
</tr>
<tr>
<td>(d : )</td>
<td>(s : )</td>
<td>(pc = p)</td>
</tr>
</tbody>
</table>

\(pc = c\)
Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical **interleaving semantics**
  - Computations of different threads are **shuffled**
  - Program order is **preserved** for each thread

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a : x = 1)</td>
<td>(p : y = 1)</td>
<td>(pc = c)</td>
</tr>
<tr>
<td>(b : \text{if}(y == 0){)</td>
<td>(q : \text{if}(x == 0){)</td>
<td>(pc = q)</td>
</tr>
<tr>
<td>(c : \text{crit. sect. 1})</td>
<td>(r : \text{crit. sect. 2})</td>
<td></td>
</tr>
<tr>
<td>(d : })</td>
<td>(s : })</td>
<td>1</td>
</tr>
</tbody>
</table>

\(pc\) = 1
Sequential Consistency (SC) Semantics [Lamport 1979]

- Threads directly write to and read from memory
- Classical **interleaving semantics**
  - Computations of different threads are shuffled
  - Program order is preserved for each thread

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
</tr>
<tr>
<td>( b : if(y == 0) )</td>
<td>( q : if(x == 0) )</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
</tr>
<tr>
<td>( d : )</td>
<td>( s : )</td>
</tr>
</tbody>
</table>

Mem

Thread 1
\( pc = c \)

Mem

Thread 2
\( pc = q \)

Mutual exclusion holds!
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- Sequential Consistency forbids compiler and hardware optimizations
- Hence is not implemented by any processor
- Processors have various buffers to reduce latency of memory accesses
- Behavior captured by relaxed memory models
- Here: Total Store Ordering (TSO) memory model
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have **write buffers**
- FIFO buffers that store writes for **later execution**
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
<td>( x )</td>
</tr>
<tr>
<td>( b : \text{if}(y == 0){ )</td>
<td>( q : \text{if}(x == 0){ )</td>
<td>( y )</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( d : } )</td>
<td>( s : } )</td>
<td></td>
</tr>
<tr>
<td>( pc = a )</td>
<td>( pc = p )</td>
<td></td>
</tr>
</tbody>
</table>
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: ( x = 1 )</td>
<td>p: ( y = 1 )</td>
<td>( w(x, 1) )</td>
</tr>
<tr>
<td>b: if(( y == 0 )){}</td>
<td>q: if(( x == 0 )){}</td>
<td></td>
</tr>
<tr>
<td>c: crit. sect. 1</td>
<td>r: crit. sect. 2</td>
<td></td>
</tr>
<tr>
<td>d: }</td>
<td>s: }</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( x )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( 0 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( y )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( 0 )</td>
</tr>
</tbody>
</table>
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have **write buffers**
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) : (x = 1)</td>
<td>(p) : (y = 1)</td>
<td>(w(x, 1))</td>
</tr>
<tr>
<td>(b) : (if(y == 0)}\</td>
<td>(q) : (if(x == 0)}\</td>
<td>(x)</td>
</tr>
<tr>
<td>(c) : (crit. sect. 1)</td>
<td>(r) : (crit. sect. 2)</td>
<td>0</td>
</tr>
<tr>
<td>(d) : }</td>
<td>(s) : }</td>
<td>(y)</td>
</tr>
</tbody>
</table>

\(w(x, 1)\)
**Total Store Ordering (TSO) Semantics [SPARC 1994, x86]**

- TSO architectures have **write buffers**
- FIFO buffers that store writes for **later execution**
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a : x = 1)</td>
<td>(p : y = 1)</td>
</tr>
<tr>
<td>(b : \text{if}(y == 0))</td>
<td>(q : \text{if}(x == 0))</td>
</tr>
<tr>
<td>(c : \text{crit. sect. 1})</td>
<td>(r : \text{crit. sect. 2})</td>
</tr>
<tr>
<td>(d : })</td>
<td>(s : })</td>
</tr>
</tbody>
</table>

Thread 1

\[pc = c\]

\[w(x, 1)\]

\[w(y, 1)\]

Mem

\(x\)

\(y\)

\(0\)

Mem

\(0\)

Roland Meyer (TU KL)  
Verification and Construction  
weapon  
May 2014  
5 / 38
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have **write buffers**
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a: x = 1)</td>
<td>(p: y = 1)</td>
<td>(w(x, 1))</td>
</tr>
<tr>
<td>(b: if(y == 0))</td>
<td>(q: if(x == 0))</td>
<td></td>
</tr>
<tr>
<td>(c: \text{crit. sect. 1})</td>
<td>(r: \text{crit. sect. 2})</td>
<td></td>
</tr>
<tr>
<td>(d: )</td>
<td>(s: )</td>
<td></td>
</tr>
</tbody>
</table>

Thread 1: \(pc = c\)

Thread 2: \(pc = q\)
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a: x = 1)</td>
<td>(p: y = 1)</td>
<td>(w(x, 1))</td>
</tr>
<tr>
<td>(b: if(y == 0))</td>
<td>(q: if(x == 0))</td>
<td>(x)</td>
</tr>
<tr>
<td>(c: crit.sect.1)</td>
<td>(r: crit.sect.2)</td>
<td>(y)</td>
</tr>
<tr>
<td>(d: )</td>
<td>(s: )</td>
<td>(1)</td>
</tr>
</tbody>
</table>
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers
- FIFO buffers that store writes for later execution
- Read takes value from memory if no write to that variable is buffered
- Otherwise read value of last write to that variable in the buffer

\[
x = y = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a: x = 1)</td>
<td>(p: y = 1)</td>
</tr>
<tr>
<td>(b: if(y == 0))</td>
<td>(q: if(x == 0))</td>
</tr>
<tr>
<td>(c: \text{crit. sect. 1})</td>
<td>(r: \text{crit. sect. 2})</td>
</tr>
<tr>
<td>(d: )</td>
<td>(s: )</td>
</tr>
</tbody>
</table>

Mem

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pc = c)</td>
<td>(pc = r)</td>
</tr>
</tbody>
</table>

\(x\) | \(1\)
\(y\) | \(1\)
TSO architectures have write buffers

FIFO buffers that store writes for later execution

Read takes value from memory if no write to that variable is buffered

Otherwise read value of last write to that variable in the buffer

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
</tr>
<tr>
<td>( b : if(y == 0) { )</td>
<td>( q : if(x == 0) { )</td>
</tr>
<tr>
<td>( c : crit. sect. 1 )</td>
<td>( r : crit. sect. 2 )</td>
</tr>
<tr>
<td>( d : } )</td>
<td>( s : } )</td>
</tr>
</tbody>
</table>

\( pc = c \)

\( pc = r \)

Mem

\( x \)

\( y \)

\( 1 \)

Mutual exclusion fails!!!
Verification Required?!

Relaxed executions may lead to bad behavior
Verification Required?!

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?
Verification Required?!

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?

Theorem [Adve, Hill 1993] If a program is data-race-free, then SC and TSO semantics coincide.
Verification Required?!

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?

**Theorem [Adve, Hill 1993]** If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!
Verification Required?! 

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?

**Theorem [Adve, Hill 1993]** If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases
Verification Required?!

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?

Theorem [Adve, Hill 1993] If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases

Performance-critical code has data races
Verification Required?!

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?

**Theorem [Adve, Hill 1993]** If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases

**Performance-critical code has data races**

Concurrency libraries  Operating systems  HPC@Fraunhofer ITWM
Verification Required?!

Relaxed executions may lead to bad behavior

If this is the real world, why does anything work?

**Theorem [Adve, Hill 1993]** If a program is data-race-free, then SC and TSO semantics coincide.

So, go and write data-race-free programs!

Works in 90% of the cases

**Performance-critical code has data races**

Concurrency libraries  Operating systems  HPC@Fraunhofer ITWM

This is where our verification techniques apply
Outline

1. **Shared Memory Concurrency**
   - Sequential Consistency Semantics
   - Total Store Ordering Semantics

2. **Reachability**

3. **Robustness**

4. **Synchronization Inference**
Reachability

[Atig, Bouajjani, Burckhardt, Musuvathi, POPL’10]
State Reachability Problem

Consider a memory model $MM$

State Reachability Problem for $MM$

**Input:** Program $P$ and a (control + memory) state $s$.

**Problem:** Is $s$ reachable when $P$ is run under $MM$?
State Reachability Problem

Consider a memory model \( MM \)

State Reachability Problem for \( MM \)

**Input:** Program \( P \) and a (control + memory) state \( s \).

**Problem:** Is \( s \) reachable when \( P \) is run under \( MM \)?

Decidability / Complexity?

Each thread is finite-state

- For the SC memory model, this problem is PSPACE-complete
State Reachability Problem

Consider a memory model $MM$

State Reachability Problem for $MM$

**Input**: Program $P$ and a (control + memory) state $s$.

**Problem**: Is $s$ reachable when $P$ is run under $MM$?

Decidability / Complexity?

Each thread is finite-state

- For the SC memory model, this problem is PSPACE-complete
- **Non-trivial** for relaxed memory models:

  $$Paths_{TSO}(P) = Closure_{TSO}(Paths_{SC}(P))$$ is non-regular
Reachability

[Atig, Bouajjani, Burckhardt, Musuvathi, POPL’10]

Decidability:

Simulation of TSO semantics by Lossy Channel Systems
Decidability of State Reachability for TSO

**Theorem [ABBM 2010]**

The *state reachability problem* for *TSO* is *reducible* to the *control-state reachability problem* for *LCS*.
The state reachability problem for TSO is reducible to the control-state reachability problem for LCS.

The control-state reachability problem for LCS is decidable.
Decidability of State Reachability for TSO

Theorem [ABBM 2010]

The state reachability problem for TSO is reducible to the control-state reachability problem for LCS.

Theorem [Abdulla, Jonsson 1993]

The control-state reachability problem for LCS is decidable.

Corollary

The state reachability problem for TSO is decidable.
Thread 1: \[ x = 1; y = 1; x = 2; y = 2; y = 3; \]

Thread 2: \[ \text{if} \ (x == 2) \{ \text{if} \ (y == 0) \{ \ldots \} \} \]

Write buffers are **perfect FIFO channels**

The write buffer of Thread 1
Thread 1: \( x = 1; y = 1; x = 2; y = 2; y = 3; \)
Thread 2: \( \text{if} (x == 2) \{ \text{if} (y == 0) \{ \ldots \} \} \)

Write buffers are perfect FIFO channels

The write buffer of Thread 1

\[
\begin{align*}
&\underbrace{w(y, 3) \ w(y, 2) \ w(x, 2) \ w(y, 1) \ w(x, 1)} \\
&\underbrace{0} \quad \underbrace{x} \quad \underbrace{0} \quad \underbrace{y}
\end{align*}
\]
From TSO to LCS  1/5

Thread 1: \( x = 1; \ y = 1; \ x = 2; \ y = 2; \ y = 3 \);

Thread 2: if \((x == 2)\) \{ if \((y == 0)\) \{ \ldots \} \}

Write buffers are perfect FIFO channels

\[
\begin{array}{c}
\text{Mem} \\
1 \\
x \\
0 \\
y \\
\end{array}
\]

\[
\begin{array}{c}
\text{w(y, 3)} \ 	ext{w(y, 2)} \ 	ext{w(x, 2)} \ 	ext{w(y, 1)} \\
\end{array}
\]

The write buffer of Thread 1
From TSO to LCS  1/5

Thread 1: \[ x = 1; \ y = 1; \ x = 2; \ y = 2; \ y = 3; \]

Thread 2: \[ \text{if} \ (x == 2) \ \{ \ \text{if} \ (y == 0) \ \{ \ldots \} \ \}\]

Write buffers are perfect FIFO channels

Write buffer of Thread 1:

\[ w(y, 3) \ w(y, 2) \ w(x, 2) \]

Mem

1
x

1
y

The write buffer of Thread 1
Thread 1: \( x = 1; \ y = 1; \ x = 2; \ y = 2; \ y = 3; \)

Thread 2: \( \text{if} (x == 2) \{ \text{if} (y == 0) \{ \ldots \} \} \)

Write buffers are **perfect FIFO channels**

\[
\begin{align*}
\text{Mem} & \\
2 & \\
\times & \\
1 & \\
y & \\
\end{align*}
\]

\[
\begin{align*}
\text{w}(y, 3) & \quad \text{w}(y, 2) \\
\text{The write buffer of Thread 1} &
\end{align*}
\]
Thread 1: $x = 1; y = 1; x = 2; y = 2; y = 3$

Thread 2: if ($x == 2$) { if ($y == 0$) { ... } }

Write buffers are perfect FIFO channels

Thread 2 reads $x = 2$
Thread 1: \[ x = 1; \ y = 1; \ x = 2; \ y = 2; \ y = 3; \]

Thread 2: \[
\text{if } (x == 2) \{ \text{if } (y == 0) \{ \ldots \} \}
\]

Write buffers are perfect FIFO channels

The write buffer of Thread 1

Thread 2 deadlocks as \( y = 1 \)
From TSO to LCS  2/5

Thread 1: \[ x = 1; \ y = 1; \ x = 2; \ y = 2; \ y = 3; \]

Thread 2: \[ \text{if } (x == 2) \{ \text{if } (y == 0) \{ \ldots \} \} \]

- Write buffers made for batch processing
- Batch processing is similar to lossiness
- So assume write buffers are lossy FIFO channels

\[
\begin{array}{c}
\text{Mem} \\
0 \\
x \\
0 \\
y \\
\end{array}
\]

\[
\begin{array}{c}
w(y, 3) \ w(y, 2) \ w(x, 2) \ w(y, 1) \ w(x, 1) \\
\end{array}
\]

The write buffer of Thread 1
Thread 1: $x = 1; y = 1; x = 2; y = 2; y = 3$

Thread 2: if $(x == 2)$ {
    if $(y == 0)$ {
        ...
    }
}

- Write buffers made for **batch processing**
- **Batch processing** is similar to **lossiness**
- So assume write buffers are **lossy** FIFO channels

![The write buffer of Thread 1](image)
From TSO to LCS 2/5

Thread 1: \( x = 1; \ y = 1; \ x = 2; \ y = 2; \ y = 3; \)

Thread 2: \( \text{if} \ (x == 2) \ \{ \ \text{if} \ (y == 0) \ \{ \ldots \} \ \} \)

- Write buffers made for **batch processing**
- **Batch processing** is similar to **lossiness**
- So assume write buffers are **lossy** FIFO channels

![Write buffer diagram]

The write buffer of Thread 1
Write buffers made for **batch processing**

**Batch processing** is similar to **lossiness**

So assume write buffers are **lossy FIFO channels**

The write buffer of Thread 1

```plaintext
w(y, 3)  w(y, 2)  w(x, 1)
```

Mem

```
0
2
x
y
```
From TSO to LCS 2/5

Thread 1: \[ x = 1; \ y = 1; \ x = 2; \ y = 2; \ y = 3; \]

Thread 2: \[ \text{if} \ (x == 2) \{ \ \text{if} \ (y == 0) \{ \ldots \} \} \}

- Write buffers made for batch processing
- Batch processing is similar to lossiness
- So assume write buffers are lossy FIFO channels

The write buffer of Thread 1

Thread 2 reads \( x = 2 \)
From TSO to LCS 2/5

Thread 1: \[ x = 1; \ y = 1; \ x = 2; \ y = 2; \ y = 3; \]

Thread 2: \[ \text{if } (x == 2) \{ \text{if } (y == 0) \{ \ldots \} \} \]

- Write buffers made for batch processing
- Batch processing is similar to lossiness
- So assume write buffers are lossy FIFO channels

Write buffers made for batch processing
Batch processing is similar to lossiness
So assume write buffers are lossy FIFO channels

\[ \begin{array}{c}
w(y, 3) \quad w(y, 2) \\
\hline
w(x, 1)
\end{array} \]

The write buffer of Thread 1

Thread 2 reads \[ y = 0 \]
From TSO to LCS 2/5

Thread 1: \( x = 1; y = 1; x = 2; y = 2; y = 3; \)

Thread 2: \( \text{if } (x == 2) \{ \text{if } (y == 0) \{ \ldots \} \} \)

- Write buffers made for **batch processing**
- **Batch processing** is similar to **lossiness**
- So assume write buffers are **lossy** FIFO channels

This is wrong! Lost the effect of \( w(y, 1) \).
From TSO to LCS  3/5

TSO buffer = perfect FIFO channel

\[
\begin{align*}
w(y, 3) & \quad w(y, 2) & w(x, 2) & w(y, 1) & w(x, 1) \\
\end{align*}
\]

Channel = sequence of memory states + lossiness

\[
\begin{align*}
x = 2 & \quad x = 2 & x = 2 & x = 1 & x = 1 \\
y = 3 & \quad y = 2 & y = 1 & y = 1 & y = 0 \\
\end{align*}
\]
From TSO to LCS  

TSO buffer = perfect FIFO channel

\[
\begin{array}{c}
\text{w}(y, 3) \quad \text{w}(y, 2) \quad \text{w}(x, 2) \quad \text{w}(y, 1) \quad \text{w}(x, 1)
\end{array}
\]

Channel = sequence of memory states + lossiness

\[
\begin{array}{c}
x = 2 \quad x = 2 \quad x = 2 \quad x = 1 \quad x = 1 \\
y = 3 \quad y = 2 \quad y = 1 \quad y = 1 \quad y = 0
\end{array}
\]

Lossiness = unobservable memory states
From TSO to LCS 3/5

TSO buffer = perfect FIFO channel

\[ w(y, 3) \ w(y, 2) \ w(x, 2) \ w(y, 1) \ w(x, 1) \]

Channel = sequence of memory states + lossiness

\[ x = 2 \quad x = 2 \quad x = 2 \quad y = 1 \quad x = 1 \quad y = 3 \quad y = 2 \quad y = 1 \quad y = 0 \]

Lossiness = unobservable memory states
From TSO to LCS  3/5

TSO buffer = perfect FIFO channel

Channel = sequence of memory states + lossiness

Lossiness = unobservable memory states
From TSO to LCS 3/5

TSO buffer = perfect FIFO channel

\[ w(y, 3) \ w(y, 2) \ w(x, 2) \ w(y, 1) \]

Channel = sequence of memory states + lossiness

\[ x = 2 \quad x = 2 \quad x = 2 \quad x = 1 \]
\[ y = 3 \quad y = 2 \quad y = 1 \quad y = 1 \]

Lossiness = unobservable memory states
From TSO to LCS  3/5

TSO buffer = perfect FIFO channel

\[ w(y, 3) \ w(y, 2) \ w(x, 2) \ w(y, 1) \]

Channel = sequence of memory states + lossiness

x = 2 \ x = 2 \ y = 1
y = 3 \ y = 2

Lossiness = unobservable memory states
From TSO to LCS 3/5

TSO buffer = perfect FIFO channel

Channel = sequence of memory states + lossiness

Lossiness = unobservable memory states
From TSO to LCS  3/5

TSO buffer = perfect FIFO channel

\[w(y, 3) \quad w(y, 2)\]

Channel = sequence of memory states + lossiness

\[\begin{align*}
x = 2 & \quad x = 2 & \quad y = 1 \\
y = 3 & \quad y = 2 & \quad y = 1
\end{align*}\]

Lossiness = unobservable memory states

Roland Meyer (TU KL)
From TSO to LCS  4/5

- **Write**: *Compute a new memory state; send it to the channel*
- **Read**: *Check the channel/memory*
- **Memory update**: *Receive a state; copy it to the memory*
Problem: Interference between threads?

- **Write**: Compute a new memory state; send it to the channel
- **Read**: Check the channel/memory
- **Memory update**: Receive a state; copy it to the memory
Problem: Interference between threads?

*Each thread guesses writes of other threads*

- **Write:** Compute a new memory state; send it to the channel
- **Read:** Check the channel/memory
- **Memory update:** Receive a state; copy it to the memory
- **Guessed Write:** Send the guessed state to the channel
Problem: Interference between threads?

Each thread guesses writes of other threads

- **Write**: Compute a new memory state; send it to the channel
- **Read**: Check the channel/memory
- **Memory update**: Receive a state; copy it to the memory
- **Guessed Write**: Send the guessed state to the channel

Check that all threads agree on their guesses
Problem: Interference between threads?

Each thread guesses writes of other threads

Thread -> Memory

- **Write**: Compute a new memory state; send it to the channel
- **Read**: Check the channel/memory
- **Memory update**: Receive a state; copy it to the memory
- **Guessed Write**: Send the guessed state to the channel

Check that all threads agree on their guesses

*Synchronization* of the LCS over send actions
Theorem [ABBM 2010]

The state reachability problem for TSO is reducible to the control-state reachability problem for LCS.
State Reachability: Conclusion

- Decidable for TSO (and beyond)
- But it is a hard problem — non-primitive recursive
- However, it is possible to have efficient analysis techniques
- Abstraction-based techniques:
  
  e.g., [Kuperstein, Vechev, Yahav, PLDI’11]

- Symbolic techniques:
  
  [Abdulla, Atig, Chen, Leonardson, Rezine, TACAS’12]
  [Linden, Wolper, SPIN’10’11]
Robustness

[Bouajjani, M., Möhlmann, ICALP’11]
[Bouajjani, Derevenetc, M., ESOP’13]
Robustness against TSO

Idea of robustness:

TSO behavior that deviates from SC is a programming error
Robustness against TSO

Idea of robustness:

TSO behavior that deviates from SC is a programming error

What is the notion of behavior?
Robustness against TSO

Idea of robustness:

TSO behavior that deviates from SC is a programming error

What is the notion of behavior?

Trace Robustness:

TSO- and SC-traces are the same [Shasha, Snir’88]
Robustness against TSO

Idea of robustness:

TSO behavior that deviates from SC is a programming error

What is the notion of behavior?

Trace Robustness:

TSO- and SC-traces are the same [Shasha, Snir’88]

Good: Allows for quite relaxed behaviors
Robustness against TSO

Idea of robustness:

TSO behavior that deviates from SC is a programming error

What is the notion of behavior?

Trace Robustness:

TSO- and SC- traces are the same [Shasha, Snir’88]

Good: Allows for quite relaxed behaviors

Very Good: Only PSPACE-complete
Traces 1/2

Computation = sequence of actions as seen by memory

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
</tr>
<tr>
<td>( b : \text{if}(y == 0)} )</td>
<td>( q : \text{if}(x == 0)} )</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
</tr>
<tr>
<td>( d : } )</td>
<td>( s : } )</td>
</tr>
</tbody>
</table>

Mem

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( pc = a )</td>
<td>( pc = p )</td>
</tr>
<tr>
<td>( x )</td>
<td>( y )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

Mem

\[ x \]
\[ 0 \]

\[ y \]
\[ 0 \]
### Traces 1/2

**Computation** = sequence of actions as seen by memory

$x = y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a : x = 1$</td>
<td>$p : y = 1$</td>
<td>$w(x, 1) \quad x$</td>
</tr>
<tr>
<td>$b : \text{if}(y == 0)$</td>
<td>$q : \text{if}(x == 0)$</td>
<td>$0$</td>
</tr>
<tr>
<td>$c : \text{crit. sect. 1}$</td>
<td>$r : \text{crit. sect. 2}$</td>
<td>$y$</td>
</tr>
<tr>
<td>$d : }$</td>
<td>$s : }$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

**Mem**

- $x$
- $0$
- $y$
- $0$
Traces  1/2

Computation = sequence of actions as seen by memory

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
<td>( )</td>
</tr>
<tr>
<td>( b : if(y == 0){ )</td>
<td>( q : if(x == 0){ )</td>
<td>( )</td>
</tr>
<tr>
<td>( c : crit. sect. 1 )</td>
<td>( r : crit. sect. 2 )</td>
<td>( )</td>
</tr>
<tr>
<td>( d : } )</td>
<td>( s : } )</td>
<td>( x )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( w(x, 1) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

\[ r(y, 0) \]
Traces 1/2

Computation = sequence of actions as seen by memory

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
<td>( x )</td>
</tr>
<tr>
<td>( b : if(y == 0){</td>
<td>( q : if(x == 0){</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
<td></td>
</tr>
<tr>
<td>( d : }</td>
<td>( s : }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( pc = c )</td>
<td>( w(x, 1) )</td>
</tr>
<tr>
<td></td>
<td>( pc = q )</td>
<td>( w(y, 1) )</td>
</tr>
</tbody>
</table>

\( r(y, 0) \)
Traces 1/2

Computation = sequence of actions as seen by memory

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a: x = 1 )</td>
<td>( p: y = 1 )</td>
</tr>
<tr>
<td>( b: if(y == 0))</td>
<td>( q: if(x == 0))</td>
</tr>
<tr>
<td>( c: \text{crit. sect. 1} )</td>
<td>( r: \text{crit. sect. 2} )</td>
</tr>
<tr>
<td>( d: )</td>
<td>( s: )</td>
</tr>
</tbody>
</table>

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

\[ Mem \]
\[ x \]
\[ 0 \]

\[ w(x, 1) \]

\[ y \]
\[ 1 \]
Traces 1/2

Computation = sequence of actions as seen by memory

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: ( x = 1 )</td>
<td>( p : y = 1 )</td>
<td>( x )</td>
</tr>
<tr>
<td>b: ( \text{if}(y == 0) { )</td>
<td>( q : \text{if}(x == 0) { )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>c: ( \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
<td>( y )</td>
</tr>
<tr>
<td>d: }</td>
<td>( s: } )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

\[ r(y, 0) \cdot w(y, 1) \cdot r(x, 0) \]
Traces 1/2

Computation = sequence of actions as seen by memory

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( b : if(y == 0) { )</td>
<td>( p : y = 1 )</td>
</tr>
<tr>
<td>( c : crit. sect. 1 )</td>
<td>( q : if(x == 0) { )</td>
<td>( r : crit. sect. 2 )</td>
</tr>
</tbody>
</table>
| \( d : \} \) | \( s : \} \) | \( s : \} \)

Thread 1 \( pc = c \)
Thread 2 \( pc = r \)

Mem

\[ r(y, 0) \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) \]
Traces 1/2

Computation = sequence of actions as seen by memory

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a : x = 1 )</td>
<td>( p : y = 1 )</td>
</tr>
<tr>
<td>( b : \text{if}(y == 0))</td>
<td>( q : \text{if}(x == 0))</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( r : \text{crit. sect. 2} )</td>
</tr>
<tr>
<td>( d : )</td>
<td>( s : )</td>
</tr>
</tbody>
</table>

Mem

\[ x \]

\[ 1 \]

\[ y \]

\[ 1 \]

\[ r(y, 0) \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) \]
Traces 2/2

Traces abstract computations to happens before dependencies

\[
\text{Trace}(r(y, 0) \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1))
\]
Traces abstract computations to happens before dependencies

- **Program order**: Order of actions issued by a thread

Trace\((r(y, 0) \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1))\)

\[\begin{align*}
  &w(x, 1) \\
  &r(y, 0) \\
  &w(y, 1) \\
  &r(x, 0)
\end{align*}\]
Traces 2/2

Traces abstract computations to happens before dependencies

- **Program order**: Order of actions issued by a thread
- **Store order**: Order of writes to a variable

Trace($r(y, 0) \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1)$)

- $w(x, 1)$
- $r(y, 0)$
- $w(y, 1)$
- $r(x, 0)$
Traces abstract computations to happens before dependencies

- **Program order**: Order of actions issued by a thread
- **Store order**: Order of writes to a variable
- **Source relation**: \textit{write} is source of \textit{read}.

Trace($r(y, 0) \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1)$)

\[
\begin{align*}
&w(x, 1) \\
&r(y, 0) \\
&w(y, 1) \\
r(x, 0)
\end{align*}
\]
Traces abstract computations to happens before dependencies

- **Program order**: Order of actions issued by a thread
- **Store order**: Order of writes to a variable
- **Source relation**: write is source of read.
- **Conflict relation**: read is overwritten by write.

\[
\text{Trace}(r(y, 0) \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1))
\]
Trace Robustness Problem

Consider a memory model $MM$

Trace Robustness Problem against $MM$

**Input:** Program $P$.

**Problem:** Does $\text{Traces}_{MM}(P) \subseteq \text{Traces}_{SC}(P)$ hold?

Decidability / Complexity?

Proof method

Theorem [Shasha, Snir 1988]

Program $P$ is robust against $MM$ iff all traces in $\text{Traces}_{MM}(P)$ are acyclic.

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

Consider a memory model $MM$

Trace Robustness Problem against $MM$

Input: Program $P$.

Problem: Does $\text{Traces}_{MM}(P) \subseteq \text{Traces}_{SC}(P)$ hold?

Decidability / Complexity?
Trace Robustness Problem

Consider a memory model $MM$

Trace Robustness Problem against $MM$

**Input:** Program $P$.

**Problem:** Does $\text{Traces}_{MM}(P) \subseteq \text{Traces}_{SC}(P)$ hold?

Decidability / Complexity?

**Proof method**

**Theorem [Shasha, Snir 1988]**

*Program $P$ is robust against MM iff all traces in $\text{Traces}_{MM}(P)$ are acyclic.*
Trace Robustness Problem

Consider a memory model \( MM \)

Trace Robustness Problem against \( MM \)

**Input:** Program \( P \).

**Problem:** Does \( \text{Traces}_{MM}(P) \subseteq \text{Traces}_{SC}(P) \) hold?

Decidability / Complexity?

Proof method

**Theorem [Shasha, Snir 1988]**

\( P \) is robust against MM iff all traces in \( \text{Traces}_{MM}(P) \) are acyclic.

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

[Bouajjani, M., Möhlmann, ICALP’11]
[Bouajjani, Derevenetc, M., ESOP’13]

Upper Bound:

Combinatorics

From Robustness to SC Reachability
Deciding Robustness

Robust Computations

Minimal Violations $= \emptyset$ ?

TSO-computations
Deciding Robustness

Robust Computations

Minimal Violations = ∅ ?

TSO-computations

Understand shape of minimal violations
Deciding Robustness

Understand shape of minimal violations

Check whether computation of this shape exists
Robustness

[Bouajjani, M., Möhlmann, ICALP’11]
[Bouajjani, Derevenetc, M., ESOP’13]

Upper Bound:

Combinatorics — Locality and Attacks

From Robustness to SC Reachability
Goal: Locality

We can restrict ourselves to violations where only one thread reorders its actions.

Proof tool: Minimal violations

Number of inversions (out-of-program-order placements) minimal among all violating computations
Locality of Robustness 2/3

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

Then $b$ and $a$ have happens before path through $\beta$
Locality of Robustness  2/3

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

Then $b$ and $a$ have happens before path through $\beta$

Subword $b_1 \ldots b_k$ with

$$b_i \rightarrow_{src/st/cf} b_{i+1} \quad \text{or} \quad b_i \rightarrow^+_p b_{i+1}$$
Consider minimal violation $\alpha \cdot b \cdot \beta \cdot a \cdot \gamma$ where $b$ has overtaken $a$

Then $b$ and $a$ have happens before path through $\beta$

Subword $b_1 \ldots b_k$ with

$$b_i \rightarrow_{src/st/cf} b_{i+1} \quad \text{or} \quad b_i \rightarrow^+_p b_{i+1}$$
Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.
Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

Pick last writes that are overtaken in two threads $t_i$ and $t_j$: 
Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

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

Case 1: No interference

---

$$
\text{Read } r_j \quad \text{Write } w_j \quad \text{Read } r_i \quad \text{Write } w_i
$$
Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

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) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

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

Case 1: No interference

Lemma: happens before cycle \( r_j \xrightarrow{+ \text{hb}} w_j \xrightarrow{+ \text{p}} r_j \)

Read \( r_i \) not involved, delete everything from \( r_i \) on
**Theorem (Locality) [BMM 2011]**

In a minimal violation, only a single thread uses its buffer.

**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
Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

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

Case 2: Overlap

\[\begin{array}{c}
r_i & \longrightarrow & r_j & \longrightarrow & w_j & \longrightarrow & w_i
\end{array}\]
Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

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  3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

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

Case 3: Interference

\[
\begin{array}{c}
\text{reading}\ r_j
\\
\text{writing}\ w_j
\\
\text{reading}\ r_i
\\
\text{writing}\ w_i
\end{array}
\]
Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

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

**Case 3: Interference**

![Diagram showing the sequence of operations](image)

Lemma: happens before cycle \( r_j \rightarrow^{+}_{hb} w_j \rightarrow^{+}_p r_j \)
Locality of Robustness  3/3

**Theorem (Locality) [BMM 2011]**

*In a minimal violation, only a single thread uses its buffer.*

**Proof sketch**

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

**Case 3: Interference**

\[ r_j \rightarrow_{hb}^{+} w_j \rightarrow_{p}^{+} r_j \]

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

Only thread $t_i$ may contribute, delete rest
Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

Proof sketch

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

Case 3: Interference

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

Only thread $t_i$ may contribute, delete rest

Lemma: happens before cycle $r_i \xrightarrow{hb} w_i \xrightarrow{p} r_i$
Locality of Robustness 3/3

Theorem (Locality) [BMM 2011]

In a minimal violation, only a single thread uses its buffer.

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
Reformulate Robustness

absence of feasible attacks
Reformulate Robustness

absence of feasible attacks

If $P$ is not robust, there are these violation:

\[ r \rightarrow \rho \rightarrow \beta \rightarrow \omega \]
Reformulate Robustness

absence of feasible attacks

If $P$ is not robust, there are these violation:

Attacker The thread that uses its buffer: only one by locality
Characterization of Robustness via Attacks  1/2

Reformulate Robustness

absence of feasible attacks

If $P$ is not robust, there are these violations:

Attacker  The thread that uses its buffer: only one by locality

Helpers  Remaining threads close cycle: $r \rightarrow_{hb}^+ w \ w \rightarrow_p^+ r$
Reformulate Robustness

absence of feasible attacks

If $P$ is not robust, there are these violation:

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

**Attacker** The thread that uses its buffer: only one by locality

**Helpers** Remaining threads close cycle: $r \xrightarrow{+}_{hb} w \xrightarrow{+}_{p} r$

\[
 r(y, 0) \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) \xrightarrow{hb}
\]
Fix thread, write instruction, read instruction

Given these parameters, find a violation as above
Characterization of Robustness via Attacks

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

**Attack**
- An attack is a triple \( A = (thread, write, read) \)
- A TSO witness for attack \( A \) is a computation as above:

\[
\alpha \rightarrow r \leftarrow \rho \rightarrow r \rightarrow \beta \rightarrow w \rightarrow \omega
\]
Fix thread, write instruction, read instruction
Given these parameters, find a violation as above

**Attack**

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

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

**Theorem [BDM’13]**

Program $P$ is robust if and only if no attack has a TSO witness.
Characterization of Robustness via Attacks  2/2

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

Attack

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

$$\alpha \quad r \quad \rho \quad r \quad \beta \quad w \quad \omega$$

Theorem [BDM’13]

Program $P$ is robust if and only if no attack has a TSO witness.

The number of attacks is quadratic in the size of $P$. 
Robustness

[Bouajjani, M., Möhlmann, ICALP’11]
[Bouajjani, Derevenetc, M., ESOP’13]

Upper Bound:

Combinatorics

From Robustness to SC Reachability
Finding TSO Witnesses with SC Reachability

TSO witnesses for attack $A$ considerably restrict TSO behavior,
Finding TSO Witnesses with SC Reachability

TSO witnesses for attack A considerably restrict TSO behavior, enough to find TSO witnesses with SC reachability.
Finding TSO Witnesses with SC Reachability

**TSO witnesses** for attack $A$ considerably restrict TSO behavior, enough to find TSO witnesses with SC reachability.

Let attacker execute under SC.
Finding TSO Witnesses with SC Reachability

**TSO witnesses** for attack A considerably restrict TSO behavior, enough to find TSO witnesses with **SC reachability**

Let attacker execute under SC

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

\[ \alpha \rightarrow w \cdot r \rightarrow \rho \rightarrow \omega \rightarrow \beta \]
Finding TSO Witnesses with SC Reachability

TSO witnesses for attack $A$ considerably restrict TSO behavior, enough to find TSO witnesses with SC reachability.

Let attacker execute under SC

Problem  Writes may conflict with helper reads

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

**TSO witnesses** for attack $A$ considerably **restrict** TSO behavior, enough to find TSO witnesses with **SC reachability**

Let attacker execute under SC

**Problem**  Writes may conflict with helper reads

**Solution**  Hide them from other threads
Finding TSO Witnesses with SC Reachability

**TSO witnesses** for attack $A$ considerably restrict TSO behavior, enough to find TSO witnesses with SC reachability.

Let attacker execute under SC

**Problem**  Writes may conflict with helper reads

**Solution**  Hide them from other threads

**Theorem [BDM’13]**

Attack $A$ has a TSO witness iff $P_A$ reaches goal state under SC.
Trace Robustness: Conclusion

- Decidable for TSO (and beyond)
- Is an easy problem — PSPACE-complete
- Locality: only one thread uses the buffer
- Analysis parallelizable

Monitoring techniques:

  e.g., [Burckhardt, Musuvathi CAV’08, Sen et al. TACAS’11]

Static analysis:

  [Shasha Snir TOPLAS’88, Alglave, Maranget CAV’11]

Semantics:

  [Owens ECOOP’10]
Synchronization Inference

[Bouajjani, Derevenetc, M., ESOP’13]
Synchronization Inference Problem

Synchronization instructions enforce visibility of commands
Synchronization Inference Problem

Synchronization instructions enforce visibility of commands
Called fences (TSO), syncs, or barriers depending on architecture
Synchronization Inference Problem

Synchronization instructions enforce visibility of commands
Called fences (TSO), syncs, or barriers depending on architecture

Consider a weak memory model $MM$

Synchronization Inference Problem for $MM$

*Input*: Program $P$ and cost function $C : \text{LAB} \rightarrow \mathbb{R}_{>0}$.  

*Problem*: Find an optimal set of synchronization instructions $F$ so that $P + F$ is robust.
Synchronization Inference Problem

Synchronization instructions enforce visibility of commands
Called fences (TSO), syncs, or barriers depending on architecture

Consider a weak memory model \( MM \)

Synchronization Inference Problem for \( MM \)

**Input**: Program \( P \) and cost function \( \mathcal{C} : \text{LAB} \to \mathbb{R}_{>0} \).

**Problem**: Find an optimal set of synchronization instructions \( F \) so that \( P + F \) is robust.

Focus on \( TSO \) (fences)
Synchronization Inference Problem

Efficiency

Fencing every write yields a robust program:

*Ruins all performance benefits brought by the memory model*
Synchronization Inference Problem

Efficiency

Fencing every write yields a robust program:

Ruins all performance benefits brought by the memory model

Solve the problem wrt. a cost function

Minimize program size or maximize program performance
Synchronization Inference Problem

Efficiency

Fencing every write yields a robust program:

*Ruins all performance benefits brought by the memory model*

Solve the problem wrt. a cost function

*Minimize program size or maximize program performance*

Decidability / Complexity / Computation

PSPACE-complete by enumeration
Synchronization Inference Problem

Efficiency

Fencing every write yields a robust program:

*Ruins all performance benefits brought by the memory model*

Solve the problem wrt. a cost function

*Minimize program size or maximize program performance*

Decidability / Complexity / Computation

PSPACE-complete by enumeration

Compute an optimal fence set

**Phase 1**: Compute candidate fence sets.

**Phase 2**: Select fence sets via integer linear programming (ILP).
Phase 1: Compute Candidate Fence Sets

**Insight 1:** Optimal fence sets are irreducible.
Phase 1: Compute Candidate Fence Sets

**Insight 1:** Optimal fence sets are irreducible.

**Insight 2:** Every irreducible fence set is a union of irreducible fence sets for all feasible attacks.
Phase 1: Compute Candidate Fence Sets

**Insight 1:** Optimal fence sets are irreducible.

**Insight 2:** Every irreducible fence set is a union of irreducible fence sets for all feasible attacks.

**Consequence:** For each attack, compute the irreducible fence sets that eliminate it.
Phase 1: Compute Candidate Fence Sets

**Insight 1:** Optimal fence sets are irreducible.

**Insight 2:** Every irreducible fence set is a union of irreducible fence sets for all feasible attacks.

**Consequence:** For each attack, compute the irreducible fence sets that eliminate it.

*This computation relies on robustness.*
Phase 2: Select an Optimal Fence Set

Assume attack $A$ has candidate fence sets $\mathcal{F}_1, \ldots, \mathcal{F}_n$. Select one:

$$\sum_{1 \leq i \leq n} x_{\mathcal{F}_i} \geq 1.$$

Make sure every location in $\mathcal{F}$ is fenced:

$$\sum_{l \in \mathcal{F}} x_l \geq |\mathcal{F}|.$$

Optimize the selected locations:

$$\sum_{l \in \text{LAB}} \mathcal{C}(x_l) \to \min.$$

Let $x^*$ be an optimal solution to the ILP problem.

Theorem

Fence set $\mathcal{F}(x^*)$ solves the synchronization inference problem.
Phase 2: Select an Optimal Fence Set

Assume attack $A$ has candidate fence sets $\mathcal{F}_1, \ldots, \mathcal{F}_n$. Select one:

$$\sum_{1 \leq i \leq n} x_{\mathcal{F}_i} \geq 1.$$  

Make sure every location in $\mathcal{F}$ is fenced:

$$\sum_{l \in \mathcal{F}} x_l \geq |\mathcal{F}| \cdot x_{\mathcal{F}}.$$
Phase 2: Select an Optimal Fence Set

Assume attack $A$ has candidate fence sets $\mathcal{F}_1, \ldots, \mathcal{F}_n$. Select one:

$$\sum_{1 \leq i \leq n} x_{\mathcal{F}_i} \geq 1.$$  

Make sure every location in $\mathcal{F}$ is fenced:

$$\sum_{l \in \mathcal{F}} x_l \geq |\mathcal{F}|x_\mathcal{F}.$$  

Optimize the selected locations:

$$\sum_{l \in \text{LAB}} c(x_l) \rightarrow \min.$$  

Let $x^*$ be an optimal solution to the ILP problem. Theorem

Fence set $\mathcal{F}(x^*)$ solves the synchronization inference problem.
Phase 2: Select an Optimal Fence Set

Assume attack $A$ has candidate fence sets $\mathcal{F}_1, \ldots, \mathcal{F}_n$. Select one:

$$\sum_{1 \leq i \leq n} x_{\mathcal{F}_i} \geq 1.$$ 

Make sure every location in $\mathcal{F}$ is fenced:

$$\sum_{l \in \mathcal{F}} x_l \geq |\mathcal{F}| x_\mathcal{F}.$$ 

Optimize the selected locations:

$$\sum_{l \in \text{LAB}} C(x_l) \rightarrow \min.$$ 

Let $x^*$ be an optimal solution to the ILP problem.

**Theorem**

Fence set $\mathcal{F}(x^*)$ solves the synchronization inference problem.