Replication and Consistency

* in and hardware general memory consistency models in particular

Roland Meyer

Technische Universität Kaiserslautern
Replication and Consistency

**Setting:** Concurrent threads accessing *shared data*
Replication and Consistency

**Setting:** Concurrent threads accessing *shared data*

Thread 1

**Shared Data**

Thread 2

Problem 1: Access to shared data is slow

Solution 1: Replicate data so that every thread has a copy
Replication and Consistency

**Setting:** Concurrent threads accessing shared data

Problem 1: Access to shared data is slow
Replication and Consistency

Setting: Concurrent threads accessing shared data

Problem 1: Access to shared data is slow
Replication and Consistency

**Setting**: Concurrent threads accessing **shared data**

**Problem 1**: Access to shared data is **slow**

**Solution 1**: Replicate data so that every thread has a **copy**
Replication and Consistency

**Setting:** Concurrent threads accessing shared data

**Problem 1:** Access to shared data is slow

**Solution 1:** Replicate data so that every thread has a copy
Replication and Consistency

Problem 2: Announce updates to other replicas
Problem 2: Announce updates to other replicas

Solution 2: Halt the system and inform everybody
Problem 2: Announce updates to other replicas

Solution 2: Halt the system and inform everybody

Ruins all performance benefits (back to Problem 1)
Replication and Consistency

**Problem 2:** Announce updates to other replicas

**Solution 2:** Halt the system and inform everybody
  - Ruins all performance benefits (back to **Problem 1**)

**Solution 2’:** Inform other threads in a delayed fashion
Replication and Consistency

Problem 2: Announce updates to other replicas

Solution 2: Halt the system and inform everybody
   Ruins all performance benefits (back to Problem 1)

Solution 2’: Inform other threads in a delayed fashion
Problem 3: Inconsistent replicas while updates travel
Problem 3: Inconsistent replicas while updates travel

State $x = 1$

Thread 1

$x = 1$

Thread 2

State $x = 0$

Solution 3: Live with it, inconsistency is here to stay!
Problem 3: Inconsistent replicas while updates travel

Solution 3:

Live with it, inconsistency is here to stay!
Replication and Consistency

Problem 3: Inconsistent replicas while updates travel

Solution 3: Live with it
Replication and Consistency

Problem 3: Inconsistent replicas while updates travel
Solution 3: Live with it

Solution 3’: Architectures give guarantees about updates
Replication and Consistency

Problem 3: Inconsistent replicas while updates travel

Solution 3: Live with it

Solution 3’: Architectures give guarantees about ordering of updates
Problem 3: Inconsistent replicas while updates travel
Solution 3: Live with it
Solution 3': Architectures give guarantees about ordering and visibility of updates
Replication and Consistency

Problem 3: Inconsistent replicas while updates travel

Solution 3: Live with it

Solution 3’: Architectures give guarantees about ordering and visibility of updates

Problem 4: But there are so many architectures
Replication and Consistency

Problem 3: Inconsistent replicas while updates travel
Solution 3: Live with it
Solution 3': Architectures give guarantees about ordering and visibility of updates

Problem 4: But there are so many architectures
Solution 4: Yes, but there are underlying principles
Replication and Consistency

Problem 3: Inconsistent replicas while updates travel
Solution 3: Live with it
Solution 3’: Architectures give guarantees about ordering and visibility of updates

Problem 4: But there are so many architectures
Solution 4: Yes, but there are underlying principles ... at least in hardware
Replication and Consistency

Principles in hardware memory consistency models
Replication and Consistency

**Principles** in hardware memory consistency models

**Guarantees** in the update mechanism [Alglave, TOPLAS’14]:
Replication and Consistency

**Principles** in *hardware memory consistency models*

**Guarantees** in the update mechanism [Alglave, TOPLAS’14]:

**SC per thread**: For one thread running in isolation the system looks consistent
Replication and Consistency

**Principles** in hardware memory consistency models

**Guarantees** in the update mechanism [Alglave, TOPLAS’14]:

**SC per thread**: For one thread running in isolation the system looks consistent

   **Consequence**: We can always rely on address and data dependencies
Replication and Consistency

**Principles** in hardware memory consistency models

** Guarantees in the update mechanism [Alglave, TOPLAS’14]:

**SC per thread:** For one thread running in isolation the system looks consistent

  **Consequence:** We can always rely on address and data dependencies

**Coherence:** For every variable all threads will see the stores to this variable in the same order
Replication and Consistency

**Principles** in hardware memory consistency models

**Guarantees** in the update mechanism [Alglave, TOPLAS’14]:

**SC per thread**: For one thread running in isolation the system looks consistent

  **Consequence**: We can always rely on address and data dependencies

**Coherence**: For every variable all threads will see the stores to this variable in the same order

**Why?** Programmability
Replication and Consistency

**Principles** in *hardware memory consistency models*

**Guarantees** in the update mechanism [Alglave, TOPLAS’14]:

**SC per thread**: For one thread running in isolation the system looks consistent

  **Consequence**: We can always rely on *address* and *data dependencies*

**Coherence**: For every variable all threads will see the stores to this variable in the same order

**Why?** Programmability + historical reasons
Replication and Consistency

Principles in hardware memory consistency models
Replication and Consistency

Principles in hardware memory consistency models

What can be relaxed:

Program order

\[ \text{TSO}^+ \text{W/R} \]
Replication and Consistency

**Principles** in hardware memory consistency models

What can be **relaxed**:

Program order

```
TSO + W/R   +W/W   PSO
```

Very strange (and not in this talk):

Out-of-thin-air values — arise when threads consistently lie to each other
Replication and Consistency

**Principles** in hardware memory consistency models

What can be **relaxed**:

Program order
Replication and Consistency

**Principles** in hardware memory consistency models

What can be relaxed:

Program order

---

Very strange (and not in this talk):
Replication and Consistency

Principles in hardware memory consistency models

What can be relaxed:

Program order

Very strange (and not in this talk):

Out-of-thin-air values — arise when threads consistently lie to each other
Replication and Consistency

Principles in hardware memory consistency models

What can be relaxed:

Program order + store order

Very strange (and not in this talk):

Out-of-thin-air values — arise when threads consistently lie to each other
Replication and Consistency

**Principles** in hardware memory consistency models

What can be *relaxed*:

Program order + store order

Very strange (and not in this talk):

**Out-of-thin-air values** — arise when threads *consistently* lie to each other
Lines of Research

Consistency models

Axiomatic, programming language (herd) for consistency models (Alglave)
Lines of Research

Consistency models
Axiomatic, programming language (herd) for consistency models (Alglave)

Geo-replicated consistency
Conflict-free replicated data types (Shapiro)
Lines of Research

Consistency models
Axiomatic, programming language (herd) for consistency models (Alglave)

Geo-replicated consistency
Conflict-free replicated data types (Shapiro)

C++11
Compilation (COMPCERT, ADVENT)
Lines of Research

Consistency models
Axiomatic, programming language (herd) for consistency models (Alglave)

Geo-replicated consistency
Conflict-free replicated data types (Shapiro)

C++11
Compilation (COMPCERT, ADVENT)

Linearizability
Semantics and algorithmics (Paderborn, Paris, Uppsala)
Lines of Research

Consistency models
Axiomatic, programming language (herd) for consistency models (Alglave)

Geo-replicated consistency
Conflict-free replicated data types (Shapiro)

C++11
Compilation (COMPCERT, ADVENT)

Linearizability
Semantics and algorithmics (Paderborn, Paris, Uppsala)

Verification under relaxed consistency models
Reachability and robustness (Paris, Uppsala, MSR, KL)
Memory Consistency Models:
TSO and SC
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 : \text{crit. sect. 1} )</td>
<td>( r : \text{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>( 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>

\[
Mem \cdot w(x, 1) \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot E
\]
### 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></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>
</tbody>
</table>

\[ isu \cdot w(x, 1) \]
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 : crit. sect. 1 )</td>
<td>( r : crit. sect. 2 )</td>
</tr>
<tr>
<td>( d : )</td>
<td>( s : )</td>
</tr>
</tbody>
</table>

Mem

\[isu \cdot w(x, 1) \cdot r(y, 0)\]
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></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>
</tbody>
</table>

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

\[
\text{isu} \cdot w(x, 1) \cdot r(y, 0) \cdot \text{isu} \cdot w(y, 1) \cdot \mathcal{E}
\]
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
TSO architectures have write buffers (FIFO)
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>

Mem:

\[ x \]
\[ y \]
\[ pc = a \]
\[ pc = p \]
\[ pc = \]
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have **write buffers** (FIFO)
- 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></td>
</tr>
<tr>
<td>(c) : <strong>crit. sect. 1</strong></td>
<td>(q) : (if(x == 0))</td>
</tr>
<tr>
<td>(d) : }</td>
<td>(r) : <strong>crit. sect. 2</strong></td>
</tr>
<tr>
<td>}</td>
<td>(s) : }</td>
</tr>
</tbody>
</table>

\(isu\)
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have **write buffers** (FIFO)
- 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><strong>a</strong>: ( x = 1 )</td>
<td><strong>p</strong>: ( y = 1 )</td>
<td></td>
</tr>
<tr>
<td><strong>b</strong>: if ( y == 0 ){}</td>
<td><strong>q</strong>: if ( x == 0 ){}</td>
<td>( x ) 0</td>
</tr>
<tr>
<td><strong>c</strong>: crit. sect. 1</td>
<td><strong>r</strong>: crit. sect. 2</td>
<td></td>
</tr>
<tr>
<td><strong>d</strong>: }</td>
<td><strong>s</strong>: }</td>
<td>( y ) 0</td>
</tr>
</tbody>
</table>

\[isu \cdot r(y, 0)\]
### Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have **write buffers** (FIFO)
- 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>

\[
\text{Thread 1}
\]

\[
\text{Thread 2}
\]

\[
\begin{align*}
\text{Mem} & \quad \text{Thread 1} & \quad \text{Thread 2} \\
0 & \quad w(x, 1) & \quad w(y, 1) \\
\end{align*}
\]

\[
isu \cdot r(y, 0) \cdot isu
\]
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers (FIFO)
- 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>( b : \text{if}(y == 0) )</td>
</tr>
<tr>
<td>( c : \text{crit. sect. 1} )</td>
<td>( q : \text{if}(x == 0) )</td>
</tr>
<tr>
<td>( d : )</td>
<td>( r : \text{crit. sect. 2} )</td>
</tr>
<tr>
<td></td>
<td>( s : )</td>
</tr>
</tbody>
</table>

\( isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \)
TSO architectures have **write buffers** (FIFO)

- 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>
<tr>
<td></td>
<td>(pc = c)</td>
</tr>
<tr>
<td></td>
<td>(pc = r)</td>
</tr>
<tr>
<td></td>
<td>(w(x, 1))</td>
</tr>
</tbody>
</table>

\[
isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0)
\]
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers (FIFO)
- 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>

Mem

\[
\text{Thread 1}
\]
\[
x
\]
\[
\text{Thread 2}
\]
\[
p = c
\]
\[
y
\]

\[
isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1)
\]
TSO architectures have write buffers (FIFO)
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

\[ x = y = 0 \]

**Thread 1**
- \[ a : x = 1 \]
- \[ b : \text{if}(y == 0)\{ \]
- \[ c : \text{crit. sect. 1} \]
- \[ d : \} \]

**Thread 2**
- \[ p : y = 1 \]
- \[ q : \text{if}(x == 0)\{ \]
- \[ r : \text{crit. sect. 2} \]
- \[ s : \} \]

\[ \text{Mutual exclusion fails!!!} \]

\[ \text{isu} \cdot r(y, 0) \cdot \text{isu} \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) \]
Total Store Ordering (TSO) Semantics [SPARC 1994, x86]

- TSO architectures have write buffers (FIFO)
- 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>

\[ \text{Mem} \]

\[ x \]

\[ y \]

\[ 1 \]

\[ \text{isu} \cdot r(y, 0) \cdot \text{isu} \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) \]
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 does have 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 does have 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 does have data races

Concurrency libraries Operating systems HPC@Fraunhofer ITWM

This is where our verification techniques apply
Reachability

[MSR, Oxford, Paris, Uppsala]
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 \textbf{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:
  $$Paths_{TSO}(P) = \text{Closure}_{TSO}(Paths_{SC}(P))$$ is non-regular
Robustness

[IMDEA, Oxford, Paris, Uppsala]

[ICALP’11, ESOP’13, ICALP’14, ACM TECS’15]
Robustness

[IMDEA, Oxford, Paris, Uppsala]
[ICALP’11, ESOP’13, ICALP’14, ACM TECS’15]

Decision procedure for robustness that
Robustness

[IMDEA, Oxford, Paris, Uppsala]

[ICALP’11, ESOP’13, ICALP’14, ACM TECS’15]

Decision procedure for robustness that

- applies to most memory models (checked TSO, PSO, PGAS, Power)
Robustness

[IMDEA, Oxford, Paris, Uppsala]

[ICALP’11, ESOP’13, ICALP’14, ACM TECS’15]

Decision procedure for robustness that

- applies to most memory models (checked TSO, PSO, PGAS, Power)
- gives precise complexity
Robustness

[IMDEA, Oxford, Paris, Uppsala]

[ICALP’11, ESOP’13, ICALP’14, ACM TECS’15]

Decision procedure for robustness that
- applies to most memory models (checked TSO, PSO, PGAS, Power)
- gives precise complexity
- ... but relies on a new automaton model and lots of guessing
Robustness

Idea: SC semantics is specification
Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error
Robustness

Idea: SC semantics is specification

- **Relaxed behavior** may contain bugs because programmers only had SC in mind
- Every **relaxed behavior** has an SC equivalent (up to traces)
- Every **relaxed behavior** that deviates from SC is a **programming error**

Robustness Problem against relaxed memory model **RMM**
Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error

Robustness Problem against relaxed memory model RMM

Input: Program $P$. 
Robustness

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error

Robustness Problem against relaxed memory model \( RMM \)

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

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

Idea: SC semantics is specification

- Relaxed behavior may contain bugs because programmers only had SC in mind
- Every relaxed behavior has an SC equivalent (up to traces)
- Every relaxed behavior that deviates from SC is a programming error

Robustness Problem against relaxed memory model \( RMM \)

\[ \text{Input: } \text{Program } P. \]

\[ \text{Problem: } \text{Does } \text{Traces}_{RMM}(P) \subseteq \text{Traces}_{SC}(P) \text{ hold?} \]

Decidability / Complexity ?
Robustness: General Solution

- Robust Computations
  - Minimal Violations $= \emptyset$?
- RMM-computations

Combinatorics: Violations can be assumed to be in normal form
Algorithmics: Check whether normal form violations exist
Robustness: General Solution

Combinatorics: Violations can be assumed to be in normal form
Robustness: General Solution

Combinatorics: Violations can be assumed to be in normal form

Algorithmics: Check whether normal form violations exist
Robustness: General Solution

Together: Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]
Robustness: General Solution

Together: Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} ? = \emptyset. \]

Combinatorics:
Robustness: General Solution

Together: Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:

- Violations to SC (if any) have a representative in normal form.
Robustness: General Solution

Together: Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} = \emptyset. \]

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:
Robustness: General Solution

Together: Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:
- Violations to SC (if any) have a representative in normal form.

Algorithmics:
- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
Robustness: General Solution

Together: Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \cap \mathcal{R}_{cyc} \) filters only \textit{violating} computations.
Robustness: General Solution

**Together:** Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

**Combinatorics:**
- Violations to SC (if any) have a representative in normal form.

**Algorithmics:**
- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \mathcal{R}_{cyc} \) filters only violating computations.
- Decide \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset \).
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \ni \emptyset. \]

Combinatorics:
- Violations to SC (if any) have a representative in normal form.

Algorithmics:
- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \cap \mathcal{R}_{cyc} \) filters only violating computations.
- Decide \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \ni \emptyset. \)
Combinatorics: Normal Form Violations

Lemma (Shasha and Snir, 1988)
A computation violates SC iff it has a cyclic happens-before relation.

\[ \tau = \text{isu} \cdot r(y, 0) \cdot \text{isu} \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) \]

Thread 1
\begin{align*}
\text{init}_x & : w(x, 1) \\
\text{init}_y & : r(y, 0)
\end{align*}

Thread 2
\begin{align*}
d & : r(x, 0) \\
c & : w(y, 1)
\end{align*}
Lemma (Shasha and Snir, 1988)
A computation violates SC iff it has a cyclic happens-before relation.

Happens-before relation of computation

\[ \tau = isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) : \]

Thread 1
- \( init_x \)
  - \( a: w(x, 1) \)

Thread 2
- \( init_y \)
  - \( b: r(y, 0) \)
  - \( d: r(x, 0) \)
  - \( c: w(y, 1) \)
Lemma (Shasha and Snir, 1988)
A computation violates SC iff it has a cyclic happens-before relation.

Happens-before relation of computation

\[ \tau = isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) : \]

- **Thread 1**
  - `init_x`
  - `a: w(x, 1)`
  - `po↓`
  - `b: r(y, 0)`

- **Thread 2**
  - `d: r(x, 0)`
  - `po↑`
  - `c: w(y, 1)`

Program order
Lemma (Shasha and Snir, 1988)
A computation violates SC iff it has a cyclic happens-before relation.

Happens-before relation of computation

\[
\tau = isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1)
\]

Program order, store order
Lemma (Shasha and Snir, 1988)
A computation violates SC iff it has a cyclic happens-before relation.

Happens-before relation of computation

\[ \tau = isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) : \]

Program order, store order, source relation
**Lemma (Shasha and Snir, 1988)**
A computation violates SC iff it has a cyclic happens-before relation.

Happens-before relation of computation

\[ \tau = isu \cdot r(y, 0) \cdot isu \cdot w(y, 1) \cdot r(x, 0) \cdot w(x, 1) : \]

Program order, store order, source relation, conflict relation
Combinatorics: Normal Form Violations

Normal Form:

\[ \tau = \tau_1 \cdot \tau_2 \]

No delays within a part

Delays in \[ \tau_2 \] respecting ordering in \[ \tau_1 \]

In normal form: ...

\[ \tau_1 \]

\[ w(x, 1) \]

\[ \tau_2 \]

Not in normal form: ...

\[ \tau_1 \]

\[ w(y, 1) \]

\[ \tau_2 \]
Combinatorics: Normal Form Violations

Normal Form:

- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
Combinatorics: Normal Form Violations

Normal Form:
- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
- No delays within a part
Combinatorics: Normal Form Violations

Normal Form:

- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
- No delays within a part
- Delays in $\tau_2$ respect ordering in $\tau_1$
**Combinatorics: Normal Form Violations**

**Normal Form:**
- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
- No delays within a part
- Delays in $\tau_2$ respect ordering in $\tau_1$

In normal form

```
. . . isu . . . isu . . . w(x, 1) . . . w(y, 1) . . .
```

\[ \tau_1 \quad \tau_2 \]
Combinatorics: Normal Form Violations

Normal Form:

- Computation has two parts $\tau = \tau_1 \cdot \tau_2$
- No delays within a part
- Delays in $\tau_2$ respect ordering in $\tau_1$

In normal form:

![Diagram showing normal form computation]

Not in normal form:

![Diagram showing non-normal form computation]
Combinatorics: Normal Form Violations

**Theorem (Normal form):**
If a program is not robust, it has a violation in normal form.
Combinatorics: Normal Form Violations

**Theorem (Normal form):**
If a program is not robust, it has a violation in normal form.

**Proof:**
- Take a *shortest* computation $\tau$ with cyclic happens-before relation.
Combinatorics: Normal Form Violations

**Theorem (Normal form):**
If a program is not robust, it has a violation in normal form.

**Proof:**
- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

\[ \tau = \tau_1 \cdot a \cdot \tau_2 . \]
Combinatorics: Normal Form Violations

**Theorem (Normal form):**
If a program is not robust, it has a violation in normal form.

**Proof:**
- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter.
Combinatorics: Normal Form Violations

Theorem (Normal form):
If a program is not robust, it has a violation in normal form.

Proof:
- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 \ .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
Combinatorics: Normal Form Violations

**Theorem (Normal form):**
If a program is not robust, it has a violation in normal form.

**Proof:**

- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:
  \[ \tau = \tau_1 \cdot a \cdot \tau_2. \]
- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
- There is an SC computation $\sigma$ with same happens-before relation.
Combinatorics: Normal Form Violations

**Theorem (Normal form):**
If a program is not robust, it has a violation in normal form.

**Proof:**
- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:
  \[ \tau = \tau_1 \cdot a \cdot \tau_2 \cdot \]
- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
- There is an SC computation $\sigma$ with same happens-before relation.
- Now
  \[ (\sigma \downarrow \tau_1) \cdot a \cdot (\sigma \downarrow \tau_2) \]
Combinatorics: Normal Form Violations

Theorem (Normal form):
If a program is not robust, it has a violation in normal form.

Proof:
- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:
  \[ \tau = \tau_1 \cdot a \cdot \tau_2. \]
- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
- There is an SC computation $\sigma$ with same happens-before relation.
- Now
  \[ (\sigma \downarrow \tau_1) \cdot a \cdot (\sigma \downarrow \tau_2) \]
  is in normal form.
Combinatorics: Normal Form Violations

Theorem (Normal form):
If a program is not robust, it has a violation in normal form.

Proof:
- Take a shortest computation $\tau$ with cyclic happens-before relation.
- There is (may be non-trivial, depending on RMM) an event that can be cancelled:

$$\tau = \tau_1 \cdot a \cdot \tau_2 .$$

- Computation $\tau_1 \cdot \tau_2$ is shorter, hence not violating.
- There is an SC computation $\sigma$ with same happens-before relation.
- Now

$$(\sigma \downarrow \tau_1) \cdot a \cdot (\sigma \downarrow \tau_2)$$

is in normal form and violating.
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} = \emptyset. \]

Combinatorics:
- Violations to SC (if any) have a representative in normal form.

Algorithmics:
- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \cap \mathcal{R}_{cyc} \) filters only violating computations.
- Decide \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} = \emptyset. \)
Algorithmics: Generating Normal-Form Computations

Challenge
Describe language $\mathcal{L}_{nf}$ of all normal-form computations
Challenge
Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that
**Challenge**

Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that

- includes $\mathcal{L}_{nf}$,
**Challenge**

Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that

- includes $\mathcal{L}_{nf}$,
- is closed under regular intersection $(\mathcal{L}_{nf} \cap \mathcal{R}_{cyc})$. 
Algorithmics: Generating Normal-Form Computations

**Challenge**

Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that

- includes $\mathcal{L}_{nf}$,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has **decidable emptiness** problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset$).
Algorithmics: Generating Normal-Form Computations

**Challenge**
Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that
- includes $\mathcal{L}_{nf}$,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has decidable emptiness problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset$).

**Properties of $\mathcal{L}_{nf}$**
**Challenge**

Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that

- includes $\mathcal{L}_{nf}$,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has decidable emptiness problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset$).

**Properties of $\mathcal{L}_{nf}$**

- Number of concurrently executed instructions is unbounded
Algorithmics: Generating Normal-Form Computations

**Challenge**
Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that
- includes $\mathcal{L}_{nf}$,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has decidable emptiness problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset$).

**Properties of $\mathcal{L}_{nf}$**
- Number of concurrently executed instructions is unbounded
- May include computations like $isu^n \cdot w(x_i, 1)^n$
**Challenge**

Describe language $\mathcal{L}_{nf}$ of all normal-form computations

Need a language class that

- includes $\mathcal{L}_{nf}$,
- is closed under regular intersection ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc}$),
- has decidable emptiness problem ($\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset$).

**Properties of $\mathcal{L}_{nf}$**

- Number of concurrently executed instructions is unbounded
- May include computations like $isu^n \cdot w(x_i, 1)^n$

  $\Rightarrow$ not context-free (language $\sigma \cdot \sigma$)
Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton
Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata

\[ \ldots isu \ldots isu \ldots \]
\[ \tau_1 \]

\[ \ldots w(x, 1) \ldots w(y, 1) \ldots \]
\[ \tau_2 \]
Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata
- Extension of NFA

\[ ... \text{isu} \ldots \text{isu} \ldots \]
\[ \tau_1 \]

\[ \ldots \text{w}(x, 1) \ldots \text{w}(y, 1) \ldots \]
\[ \tau_2 \]
Algorithmics: Generating Normal-Form Computations

Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata

- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously

\[ \ldots \text{isu} \ldots \text{isu} \ldots \]
\[ \tau_1 \]
\[ \ldots \text{w}(x, 1) \ldots \text{w}(y, 1) \ldots \]
\[ \tau_2 \]
Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata

- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$
Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata
- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:
Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata
- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:
Solution
Define $\mathcal{L}_{nf}$ as language of a **multiheaded automaton**

**Multiheaded automata**
- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:
Solution
Define \( \mathcal{L}_{nf} \) as language of a multiheaded automaton

Multiheaded automata
- Extension of NFA
- Generates parts \( \tau_1 \) and \( \tau_2 \) of a computation \( \tau_1 \cdot \tau_2 \) simultaneously
- Transitions \( q \xrightarrow{1,a} q' \) and \( q \xrightarrow{2,b} q' \) labeled by head \( i = 1, 2 \)

Example:

\[
\begin{align*}
\ldots \uparrow isu \ldots isu \ldots & \quad \ldots \uparrow w(x, 1) \ldots w(y, 1) \\
\tau_1 & \quad \tau_2
\end{align*}
\]
Solution
Define $L_{nf}$ as language of a multiheaded automaton

Multiheaded automata
- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:

Transitions: $q_1 \xrightarrow{1,isu} q_2 \xrightarrow{2,w(x,1)} q_3$
Solution
Define $\mathcal{L}_{nf}$ as language of a multiheaded automaton

Multiheaded automata
- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:

Transitions: $q_1 \xrightarrow{1,isu} q_2 \xrightarrow{2,w(x,1)} q_3$
Solution
Define $\mathcal{L}_{nf}$ as language of a **multiheaded automaton**

**Multiheaded automata**
- Extension of NFA
- Generates parts $\tau_1$ and $\tau_2$ of a computation $\tau_1 \cdot \tau_2$ simultaneously
- Transitions $q \xrightarrow{1,a} q'$ and $q \xrightarrow{2,b} q'$ labeled by head $i = 1, 2$

Example:

Transitions: $q_1 \xrightarrow{1,isu} q_2 \xrightarrow{2,w(x,1)} q_3$
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} = \emptyset. \]

Combinatorics:

- Violations to SC (if any) have a representative in normal form.

Algorithmics:

- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \cap \mathcal{R}_{cyc} \) filters only violating computations.
- Decide \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} = \emptyset. \)
Algorithmics: Checking Cyclicity

Happens-before relation from the example:

\[
\begin{align*}
\text{Thread 1} & \\
\text{init}_x & \\
& \xrightarrow{\text{st}} \quad a: \ w(x, 1) \\
& \quad \xrightarrow{\text{po}} \\
\text{Thread 2} & \\
d: \ r(x, 0) & \\
& \xleftarrow{\text{cf}} \quad c: \ w(y, 1) \\
& \quad \xleftarrow{\text{cf}} \\
\text{Thread 1} & \\
b: \ r(y, 0) & \\
& \xrightarrow{\text{st}} \\
\text{Thread 2} & \\
d: \ r(x, 0) & \\
& \xleftarrow{\text{cf}} \quad c: \ w(y, 1) \\
& \quad \xleftarrow{\text{cf}} \\
\end{align*}
\]
Happens-before relation from the example:

```
Thread 1
init_x
  st
  src
  po
  cf
  st

Thread 2
init_y
  src
  po
  cf

Thread 1
  a: w(x, 1)
  b: r(y, 0)

Thread 2
  d: r(x, 0)
  c: w(y, 1)
```

Checking cyclicity
Algorithmics: Checking Cyclicity

Happens-before relation from the example:

Checking cyclicity
- Finitely many types of cycles
Algorithmics: Checking Cyclicity

Happens-before relation from the example:

```
Thread 1
init_x → a: w(x, 1)

Thread 2
src → d: r(x, 0)

init_y → src → b: r(y, 0)

src → c: w(y, 1)
```

Checking cyclicity

- Finitely many types of cycles
- **Guess** per thread two instructions in program order
Algorithmics: Checking Cyclicity

Happens-before relation from the example:

- **Thread 1**
  - `init_x`: `w(x, 1)`
  - `a`: `w(x, 1)`
  - `b`: `r(y, 0)`
  - `c`: `w(y, 1)`

- **Thread 2**
  - `d`: `r(x, 0)`

checking cyclicity

- Finitely many types of cycles
- Guess per thread two instructions in program order
- **Finite automata check edges** between guessed instructions from different threads
Robustness: General Solution

Reduce robustness to an emptiness check

\[ \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset. \]

Combinatorics:
- Violations to SC (if any) have a representative in normal form.

Algorithmics:
- Language \( \mathcal{L}_{nf} \) consists of all normal-form computations.
- \( \cap \mathcal{R}_{cyc} \) filters only violating computations.
- Decide \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset \).
Algorithmics: Emptiness

**Theorem:**
Assuming finite memory, robustness is $\text{PSPACE}$-complete.
Algorithmics: Emptiness

Theorem:
Assuming finite memory, robustness is $PSPACE$-complete.

Proof:
Theorem:
Assuming finite memory, robustness is $\text{PSPACE}$-complete.

Proof:
- Upper bound: $\mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset$.
Algorithmics: Emptiness

Theorem:
Assuming finite memory, robustness is \( \text{PSPACE}-\text{complete} \).

Proof:
- Upper bound: \( \mathcal{L}_{nf} \cap \mathcal{R}_{cyc} \neq \emptyset \).
- Lower bound: SC state reachability [Kozen 1977].