Lecture 3: Verification of Weak Memory Models

Part 1: State Reachability Problem

Ahmed Bouajjani

LIAFA, University Paris Diderot – Paris 7

[Atig, B., Burckhardt, Musuvathi, POPL’10, ESOP’12]
[Atig, B., Parlato, 2011]

VTSA, MPI-Saarbrücken, September 2012
Sequential Consistency (SC) model

- Parallel processes with shared memory

- Interleaving (Sequentially Consistent) semantics:
  - Computations of different processes are shuffled
  - Program order is preserved for each process.
Total Store Ordering (TSO)

- Reads can overtake writes on $\neq$ variables.
- FIFO buffers where writes are stored to be executed later.
- Reads take values from the main memory if no writes in the buffer on the same variable. Otherwise they get the value of the last write in the buffer on the same variable.
Write-to-Read Relaxation

\[ P_1 : \text{write}(x, 1) ; \text{read}(y, 0) \]
\[ P_2 : \text{read}(x, 0) \]

A scheduling for SC semantics: 3 steps

\[ P_1 : \text{write}(x, 1)_2 ; \text{read}(y, 0)_3 \]
\[ P_2 : \text{read}(x, 0)_1 \]
Write-to-Read Relaxation

\[ P_1 : \text{write}(x, 1) \; ; \; \text{read}(y, 0) \]
\[ P_2 : \text{read}(x, 0) \]

A scheduling for SC semantics: 3 steps

\[ P_1 : \text{write}(x, 1)_{(2)} \; ; \; \text{read}(y, 0)_{(3)} \]
\[ P_2 : \text{read}(x, 0)_{(1)} \]

Allowing reordering of actions on different variables: 2 steps!

\[ P_1 : \text{read}(y, 0)_{(1)} \; ; \; \text{write}(x, 1)_{(2)} \]
\[ P_2 : \text{read}(x, 0)_{(1)} \]
Relaxed Models

- **Read Local Write Early**
  \[ \text{write} \ (x,d) \ ; \ \text{read} \ (x,d) \mapsto \text{write} \ (x,d) \]

- **(+) \ W \rightarrow R:** Write to Read
  \[ \text{write} \ (x,d) \ ; \ \text{read} \ (y,d') \mapsto \text{read} \ (y,d') \ ; \ \text{write} \ (x,d) \]

  \[ \Rightarrow \text{TSO model} \ (\text{Total Store Ordering}) \]

- **(+) \ W \rightarrow W:** Write to Write
  \[ \text{write} \ (x,d) \ ; \ \text{write} \ (y,d') \mapsto \text{write} \ (y,d') \ ; \ \text{write} \ (x,d) \]

  \[ \Rightarrow \text{PSO model} \ (\text{Partial Store Ordering}) \]

- **(+) \ R \rightarrow R/W:** Read to Read/Write
  \[ \Rightarrow \sim\text{RMO model} \ (\text{Relaxed Memory Ordering}) \]
Relaxation ⇒ Potential Bad Behaviors

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: ( y = 1 )</td>
<td>p: ( x = 1 )</td>
</tr>
<tr>
<td>b: ( r_1 = x )</td>
<td>q: ( r_2 = y )</td>
</tr>
<tr>
<td>c: if(( r_1 == 0 )) {</td>
<td>s: if(( r_2 == 0 )) {</td>
</tr>
<tr>
<td>d: ...</td>
<td>t: ...</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

1- Initial state

\[ pc_1 = a \]
\[ r_1 = ? \]
\[ pc_2 = p \]
\[ r_2 = ? \]

shared memory

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

Dekker’s mutual exclusion protocol. Fails under Write to Read relaxation.
Dekker's mutual exclusion protocol. Fails under Write to Read relaxation.
Relaxation ⇒ Potential Bad Behaviors

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: ( y = 1 )</td>
<td>p: ( x = 1 )</td>
</tr>
<tr>
<td>b: ( r_1 = x )</td>
<td>q: ( r_2 = y )</td>
</tr>
<tr>
<td>c: if(( r_1 == 0 )) {</td>
<td>s: if(( r_2 == 0 )) {</td>
</tr>
<tr>
<td>d: ...</td>
<td>t: ...</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

3- Reading from memory

\[
\begin{align*}
\text{thread 1} & \quad \text{thread 2} \\
pc_1 &= c & pc_2 &= s \\
r_1 &= 0 & r_2 &= 0 \\
w(y, 1) & & w(x, 1) \\
\text{shared memory} & & \\
x &= 0 & y &= 0
\end{align*}
\]

Dekker's mutual exclusion protocol. Fails under Write to Read relaxation.
Relaxation ⇒ Potential Bad Behaviors

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: ( y = 1 )</td>
<td>p: ( x = 1 )</td>
</tr>
<tr>
<td>b: ( r_1 = x )</td>
<td>q: ( r_2 = y )</td>
</tr>
<tr>
<td>c: if(( r_1 == 0 )) {</td>
<td>s: if(( r_2 == 0 )) {</td>
</tr>
<tr>
<td>d: ...</td>
<td>t: ...</td>
</tr>
</tbody>
</table>
| } }               | } }

4- Accessing critical sections

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( pc_1 = d )</td>
<td>( pc_2 = t )</td>
</tr>
<tr>
<td>( r_1 = 0 )</td>
<td>( r_2 = 0 )</td>
</tr>
<tr>
<td>[ w(y, 1) ]</td>
<td>[ w(x, 1) ]</td>
</tr>
</tbody>
</table>

shared memory

\[ x = 0 \quad y = 0 \]

Dekker’s mutual exclusion protocol. Fails under Write to Read relaxation.
Memory Reordering Fences

- **Write-Write Fences (wfence):**
  
  \textit{Prevent reordering between writes.}

- **Read-Read Fences (rfence):**

  \textit{Prevent reordering between reads.}

- **Fences (fence):**

  \textit{Prevent reordering between any two memory operations.}
Program Syntax

- Finite number of shared variables \( \{x, y, x_1, \ldots\} \)
- Finite data domain \( \{d, d_1, d_2, \ldots\} \)
- Finite number of finite-control processes \( P_1, \ldots, P_n \) with operations:
  
  \[
  \text{Write}(x, d), \text{Wfence}, \text{Read}(x, d), \text{Rfence}, \text{AtomicRW}(x, d_1, d_2)
  \]
Safety Verification Problem

For a memory model $\mu$, a program $P$, and a (control + memory) state $s$

- State Reachability Problem (Safety)
  
  $s$ is reachable in $P$ ?
Safety Verification Problem

For a memory model $\mu$, a program $P$, and a (control + memory) state $s$

- State Reachability Problem (Safety)

  $s$ is reachable in $P$ ?

Decidability / Complexity ?

Each process is finite-state

- For the SC memory model, this problem is PSPACE-complete
Safety Verification Problem

For a memory model $\mu$, a program $P$, and a (control + memory) state $s$

- State Reachability Problem (Safety)
  
  $s$ is reachable in $P$ ?

Decidability / Complexity ?

Each process is finite-state

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

- Nontrivial for weak memory models:

\[ \text{Paths}_\mu(P) = \text{Closure}_\mu(\text{Paths}_{\text{SC}}(P)) \text{ is nonregular} \]
Results for TSO [Atig, B., Burckhardt, Musuvathi, 2010]

- The state reachability problem is **decidable** for TSO.
Results for TSO [Atig, B., Burckhardt, Musuvathi, 2010]

- The state reachability problem is *decidable* for TSO.
- ... but highly complex: Nonprimitive recursive
Results for TSO [Atig, B., Burckhardt, Musuvathi, 2010]

- The state reachability problem is decidable for TSO.
- ... but highly complex: Nonprimitive recursive
- The repeated state reachability problem is undecidable for TSO
The state reachability problem is decidable for TSO.
... but highly complex: Nonprimitive recursive

The repeated state reachability problem is undecidable for TSO

→ Store buffers can simulate lossy channels, and vice-versa.
The state reachability problem is **undecidable** for

\[ TSO + R2W \]
Decidability Frontier [Atig, B., Burckhardt, Musuvathi, 2012]

- The state reachability problem is **undecidable** for
  \( TSO + R2W \)

- The state reachability problem is **decidable** for
  \( NSW = TSO + W2W + R2R \)
Getting rid of Store Buffers [Atig, B., Parlato, 2011]

- When is it possible to reduce TSO verification to SC verification?
Getting rid of Store Buffers [Atig, B., Parlato, 2011]

- When is it possible to **reduce TSO verification to SC verification**?

- **Find restrictions** on the explored behaviors such that:

  Given a concurrent program $P$, it is possible to build a concurrent program $P'$ such that: running $P$ with TSO semantics under these restrictions is equivalent to running $P'$ with SC semantics.
Getting rid of Store Buffers [Atig, B., Parlato, 2011]

- When is it possible to reduce TSO verification to SC verification?
- Find restrictions on the explored behaviors such that:
  
  Given a concurrent program $P$, it is possible to build a concurrent program $P'$ such that: running $P$ with TSO semantics under these restrictions is equivalent to running $P'$ with SC semantics.

- A notion of Context-Bounded Analysis for TSO
Getting rid of Store Buffers [Atig, B., Parlato, 2011]

- When is it possible to reduce TSO verification to SC verification?

- **Find restrictions** on the explored behaviors such that:

  > Given a concurrent program $P$, it is possible to build a concurrent program $P'$ such that: running $P$ with TSO semantics under these restrictions is equivalent to running $P'$ with SC semantics.

- A notion of **Context-Bounded Analysis for TSO**

- Unbounded number of context-switches: Bounding the age of each write in the buffer in terms of context-switches.
Getting rid of Store Buffers [Atig, B., Parlato, 2011]

- When is it possible to reduce TSO verification to SC verification?
- Find restrictions on the explored behaviors such that:
  
  \[\text{Given a concurrent program } P, \text{ it is possible to build a concurrent program } P' \text{ such that: running } P \text{ with TSO semantics under these restrictions is equivalent to running } P' \text{ with SC semantics.}\]

- A notion of Context-Bounded Analysis for TSO

- Unbounded number of context-switches: Bounding the age of each write in the buffer in terms of context-switches.
  
  \[\Rightarrow \text{ Transfer decidability/complexity results from SC to TSO.}\]
  
  \[\Rightarrow \text{ Use existing tools for concurrent programs under SC.}\]
The rest of the lecture

- Decidability and complexity for TSO:
  
  *Simulations by/of* Lossy Channel Systems
The rest of the lecture

- **Decidability and complexity for TSO:**
  
  *Simulations by/of Lossy Channel Systems*

- **Decidability and complexity beyond TSO:**
  
  - Speculative writes lead to undecidability
  - Decidability: deal with reordered reads
The rest of the lecture

- Decidability and complexity for TSO:
  
  *Simulations by/of Lossy Channel Systems*

- Decidability and complexity beyond TSO:
  
  - Speculative writes lead to undecidability
  - Decidability: deal with reordered reads

- From TSO to SC under bounded analysis
  
  - 2 notions of bounds
  - *Store buffers* $\leadsto$ 2K copies of the globals per thread
An operational model for TSO

- Each process has a FIFO buffer
- Configuration = control states + memory state + buffers contents
- Write($x,d$) is sent to the buffer
- Memory update = execution of a Write taken from some buffer
- Read($x,d$) is executed either if
  - The last Write to $x$ in the buffer is Write($x,d$) (Read Own Write)
  - The buffer does not contain a Write to $x$, and Memory($x$) = $d$
- AtomicRW($x, d_1, d_2$) requires that the buffer is empty ($\sim$ fence)
From \( W \rightarrow R \) systems to Lossy Channel Systems

**Thread 1:**

1. \( p_0 \xrightarrow{w(x, 1)} p_1 \)
2. \( p_1 \xrightarrow{w(y, 1)} p_2 \)
3. \( p_2 \xrightarrow{w(x, 2)} p_3 \)
4. \( p_3 \xrightarrow{w(y, 2)} p_4 \)
5. \( p_4 \xrightarrow{w(y, 3)} p_5 \)

**Thread 2:**

1. \( q_0 \xrightarrow{r(x, 2)} q_1 \)
2. \( q_1 \xrightarrow{r(y, 0)} q_2 \)

**Model:** The store buffers are considered as perfect FIFO channels

---

The store buffer of Thread 1
From W → R systems to Lossy Channel Systems

Thread 1: $p_0 \xrightarrow{w(x,1)} p_1 \xrightarrow{w(y,1)} p_2 \xrightarrow{w(x,2)} p_3 \xrightarrow{w(y,2)} p_4 \xrightarrow{w(y,3)} p_5$

Thread 2: $q_0 \xrightarrow{r(x,2)} q_1 \xrightarrow{r(y,0)} q_2$

Model: The store buffers are considered as perfect FIFO channels

The store buffer of Thread 1

\[ w(y,3) \ x \ w(y,2) \ w(x,2) \ w(y,1) \ w(x,1) \]
From $W \rightarrow R$ systems to Lossy Channel Systems

Thread 1:

\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{w(x, 2)} p_3 \xrightarrow{w(y, 2)} p_4 \xrightarrow{w(y, 3)} p_5 \]

Thread 2:

\[ q_0 \xrightarrow{r(x, 2)} q_1 \xrightarrow{r(y, 0)} q_2 \]

Model: The store buffers are considered as perfect FIFO channels

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

The store buffer of Thread 1

1

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

The store buffer of Thread 1

0

y
From $W$ → $R$ systems to Lossy Channel Systems

Thread 1: $p_0$ $w(x, 1)$ $p_1$ $w(y, 1)$ $p_2$ $w(x, 2)$ $p_3$ $w(y, 2)$ $p_4$ $w(y, 3)$ $p_5$

Thread 2: $q_0$ $r(x, 2)$ $q_1$ $r(y, 0)$ $q_2$

Model: The store buffers are considered as perfect FIFO channels

---

The store buffer of Thread 1

$w(y, 3)$ $w(y, 2)$ $w(x, 2)$
From \( W \to R \) systems to Lossy Channel Systems

Thread 1:

\[
\begin{array}{c}
p_0 \\ w(x, 1)\\ p_1 \\ w(y, 1) \\ p_2 \\ w(x, 2) \\ p_3 \\ w(y, 2) \\ p_4 \\ w(y, 3) \\ p_5
\end{array}
\]

Thread 2:

\[
\begin{array}{c}
q_0 \\ r(x, 2)\\ q_1 \\ r(y, 0)\\ q_2
\end{array}
\]

Model: The store buffers are considered as perfect FIFO channels

---

2

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

1

x

y

The store buffer of Thread 1
From $W \rightarrow R$ systems to Lossy Channel Systems

Thread 1: $p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{w(x, 2)} p_3 \xrightarrow{w(y, 2)} p_4 \xrightarrow{w(y, 3)} p_5$

Thread 2: $q_0 \xrightarrow{r(x, 2)} q_1 \xrightarrow{r(y, 0)} q_2$

Model: The store buffers are considered as perfect FIFO channels

---

The store buffer of Thread 1
From $W \rightarrow R$ systems to Lossy Channel Systems

Thread 1: $p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{w(x, 2)} p_3 \xrightarrow{w(y, 2)} p_4 \xrightarrow{w(y, 3)} p_5$

Thread 2: $q_0 \xrightarrow{r(x, 2)} q_1 \xrightarrow{r(y, 0)} q_2$

Model: The store buffers are considered as perfect FIFO channels

Deadlock

The store buffer of Thread 1
From $W \rightarrow R$ systems to Lossy Channel Systems

Thread 1:

\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{w(x, 2)} p_3 \xrightarrow{w(y, 2)} p_4 \xrightarrow{w(y, 3)} p_5 \]

Thread 2:

\[ q_0 \xrightarrow{r(x, 2)} q_1 \xrightarrow{r(y, 0)} q_2 \]

Assume that the store buffers are **lossy** FIFO channels

### The store buffer of Thread 1

\[
\begin{array}{cccccccc}
w(y, 3) & w(y, 2) & w(x, 2) & w(y, 1) & w(x, 1) \\
\end{array}
\]
From \( W \rightarrow R \) systems to Lossy Channel Systems

Thread 1:

\[
\begin{align*}
p_0 & \xrightarrow{w(x,1)} p_1 \xrightarrow{w(y,1)} p_2 \xrightarrow{w(x,2)} p_3 \xrightarrow{w(y,2)} p_4 \xrightarrow{w(y,3)} p_5
\end{align*}
\]

Thread 2:

\[
\begin{align*}
q_0 & \xrightarrow{r(x,2)} q_1 \xrightarrow{r(y,0)} q_2
\end{align*}
\]

Assume that the store buffers are lossy FIFO channels.

The store buffer of Thread 1
From $W \rightarrow R$ systems to Lossy Channel Systems

Thread 1: 

$$w(x, 1) \rightarrow p_1 \rightarrow w(y, 1) \rightarrow p_2 \rightarrow w(x, 2) \rightarrow p_3 \rightarrow w(y, 2) \rightarrow p_4 \rightarrow w(y, 3) \rightarrow p_5$$

Thread 2: 

$$r(x, 2) \rightarrow q_1 \rightarrow r(y, 0) \rightarrow q_2$$

Assume that the store buffers are **lossy** FIFO channels

The store buffer of Thread 1

\[w(y, 3) \ x \ y\]
From $W \rightarrow R$ systems to Lossy Channel Systems

**Thread 1:**

1. $p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{w(x, 2)} p_3 \xrightarrow{w(y, 2)} p_4 \xrightarrow{w(y, 3)} p_5$

**Thread 2:**

1. $q_0 \xrightarrow{r(x, 2)} q_1 \xrightarrow{r(y, 0)} q_2$

Assume that the store buffers are **lossy** FIFO channels
From $W \to R$ systems to Lossy Channel Systems

Thread 1: $p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{w(x, 2)} p_3 \xrightarrow{w(y, 2)} p_4 \xrightarrow{w(y, 3)} p_5$

Thread 2: $q_0 \xrightarrow{r(x, 2)} q_1 \xrightarrow{r(y, 0)} q_2$

Assume that the store buffers are lossy FIFO channels

The store buffer of Thread 1
From W → R systems to Lossy Channel Systems

Thread 1:  
\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{w(x, 2)} p_3 \xrightarrow{w(y, 2)} p_4 \xrightarrow{w(y, 3)} p_5 \]

Thread 2:  
\[ q_0 \xrightarrow{r(x, 2)} q_1 \xrightarrow{r(y, 0)} q_2 \]

Assume that the store buffers are lossy FIFO channels
From $W \rightarrow R$ systems to Lossy Channel Systems

Buffer = perfect FIFO channel

Channel = Sequence of memory states + Lossyness
From $W \rightarrow R$ systems to Lossy Channel Systems

Buffer $= \text{perfect FIFO channel}$

Channel $= \text{Sequence of memory states} + \text{Lossyness}$

Lossyness $= \text{Unobservable memory states}$
From $W \rightarrow R$ systems to Lossy Channel Systems

**Buffer** = perfect FIFO channel

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

**Channel** = Sequence of memory states + Lossyness

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

**Lossyness** = Unobservable memory states
From $W \rightarrow R$ systems to Lossy Channel Systems

Buffer = perfect FIFO channel

Channel = Sequence of memory states + Lossyness

Lossyness = Unobservable memory states
From $W \to R$ systems to Lossy Channel Systems

Buffer = perfect FIFO channel

Channel = Sequence of memory states + Lossyness

Lossyness = Unobservable memory states
From $W \rightarrow R$ systems to Lossy Channel Systems

Buffer = perfect FIFO channel

Channel = Sequence of memory states + Lossyness

Lossyness = Unobservable memory states
From $W \rightarrow R$ systems to Lossy Channel Systems

Buffer = perfect FIFO channel

Channel = Sequence of memory states + Lossyness

Lossyness = Unobservable memory states
From $W \rightarrow R$ systems to Lossy Channel Systems

Buffer = perfect FIFO channel

Channel = Sequence of memory states + Lossyness

Lossyness = Unobservable memory states
From W → R systems to Lossy Channel Systems

- **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 processes?

- **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
From $W \rightarrow R$ systems to Lossy Channel Systems

- **Problem:** Interference between processes?
- $\Rightarrow$ Each process guesses occurrences of writes by other processes

---

**Process** $\rightarrow$ 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*
From $W \rightarrow R$ systems to Lossy Channel Systems

- **Problem:** Interference between processes?

$\Rightarrow$ Each process guesses occurrences of writes by other processes

- **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

$\Rightarrow$ Check that all processes agree on the sequence of states

*Synchronization of the lossy channel machines over send actions*
Decidability for the State Reachability Problem

- **Thm**

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

\[ \text{Thm} \]

\textit{The state reachability problem for TSO programs is reducible to the control-state reachability problem for LCS.}
Decidability for the State Reachability Problem

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

- **Thm ([Abdulla, Jonsson, 1993])**
  
  *The control-state reachability problem for LCS is decidable*

- **Corollary**
  
  *The state reachability problem for TSO systems is decidable.*
From Lossy Channel Systems to W → R systems

- \( T_1 \) simulates the lossy channel machine:
  - Send operation: Write operation of \( T_1 \) to the variable \( x \)
  - Read operation: Read operation of \( T_1 \) from the variable \( y \)

- \( T_2 \) transfers the successive values of the variable \( x \) to the variable \( y \)
Complexity

- **Thm**

\[\text{Every LCS can be simulated by a TSO program.}\]
Complexity

- **Thm**
  
  *Every LCS can be simulated by a TSO program.*

- **Thm** ([Schnoebelen, 2001])
  
  *The control-state reachability problem for LCS is non-primitive recursive*

  $\Rightarrow$ Lower bound for the state reachability problem under TSO.
TSO + R2W: Causality cycles

The behavior is possible since writes can overtake reads:

(2), (3), (4), (1)

Speculative writes \[ \Rightarrow \] causality cycles

(2) is executed assuming that (1) will be executed in the future

(1) is indeed executed, but it is based on a write that depends from (2)

<table>
<thead>
<tr>
<th>( P_1 )</th>
<th>( P_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) ( r(x, 1) )</td>
<td>(3) ( r(y, 1) )</td>
</tr>
<tr>
<td>(2) ( w(y, 1) )</td>
<td>(4) ( w(x, 1) )</td>
</tr>
</tbody>
</table>

\( x = y = 0 \)

\( x = y = 1 \)
TSO + R2W: Causality cycles

\[ x = y = 0 \]

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) $r(x, 1)$</td>
<td>(3) $r(y, 1)$</td>
</tr>
<tr>
<td>(2) $w(y, 1)$</td>
<td>(4) $w(x, 1)$</td>
</tr>
</tbody>
</table>

\[ x = y = 1 \]

- This behavior is possible since writes can overtake reads:
  (2), (3), (4), (1)

- Speculative writes $\Rightarrow$ causality cycles
  - (2) is executed assuming that (1) will be executed in the future
  - (1) is indeed executed, but it is based on a write that depends from (2)
TSO + R2W: Undecidability

Assume that: $u_{i_1} u_{i_2} \cdots u_{i_n} = v_{j_1} v_{j_2} \cdots v_{j_m}$ and $i_1 i_2 \cdots i_n = j_1 j_2 \cdots j_m$

$T_1: \ r(y_2, i_n) \ w(y_1, i_n) \ r(x_2, u_{i_n}) \ w(x_1, u_{i_n}) \ \cdots \ r(y_2, i_1) \ w(y_1, i_1) \ r(x_2, u_{i_1}) \ w(x_1, u_{i_1})$

$T_2: \ r(y_1, j_n) \ w(y_2, j_n) \ r(x_1, v_{j_n}) \ w(x_2, v_{j_n}) \ \cdots \ r(y_1, j_1) \ w(y_2, j_1) \ r(x_1, v_{j_1}) \ w(x_2, v_{j_1})$
TSO + R2W: Undecidability

Assume that: \( u_{i_1} u_{i_2} \cdots u_{i_n} = v_{j_1} v_{j_2} \cdots v_{j_m} \) and \( i_1 i_2 \cdots i_n = j_1 j_2 \cdots j_m \)

\[ T_1: w(x_2, v_{i_1}) \cdots w(x_2, v_{i_n}) \cdots w(x_2, u_{i_1}) \cdots w(y_2, u_{i_n}) \cdots w(y_2, i_1) \cdots w(x_1, i_n) \cdots w(x_1, u_{i_1}) \]

\[ T_2: w(y_2, j_n) \cdots w(y_2, v_{j_n}) \cdots w(y_2, j_1) \cdots w(x_2, v_{j_1}) \cdots w(y_1, j_n) \cdots w(x_1, v_{j_n}) \cdots w(y_1, j_1) \cdots w(x_1, v_{j_1}) \]
Assume that: \( u_{i_1} u_{i_2} \cdots u_{i_n} = v_{j_1} v_{j_2} \cdots v_{j_m} \) and \( i_1 i_2 \cdots i_n = j_1 j_2 \cdots j_m \)

\[ T_1: r(y_2, i_n) \ r(x_2, u_{i_n}) \ \cdots \ r(y_2, i_1) \ r(x_2, u_{i_1}) \ \cdots \ w(y_1, i_n) \ w(x_1, u_{i_n}) \ \cdots \ w(y_1, i_1) \ w(x_1, u_{i_1}) \]

\[ T_2: w(y_2, j_n) \ w(x_2, v_{j_n}) \ \cdots \ w(y_2, j_1) \ w(x_2, v_{j_1}) \ \cdots \ r(y_1, j_n) \ r(x_1, v_{j_n}) \ \cdots \ r(y_1, j_1) \ r(x_1, v_{j_1}) \]

\[ \Rightarrow \text{Reachability TSO + R2W} \]
NSW: Non Speculative Writes

- TSO = Read-Local-Write-Early + W2R
- PSO = TSO + W2W
- NSW = PSO + R2R

Simulation of TSO under PSO:
Add a write-write fence (\textit{wfence}) before each write

Simulation of PSO under NSW:
Add a read-read fence (\textit{rfence}) before each read
NSW: Non Speculative Writes

- TSO = Read-Local-Write-Early + W2R
- PSO = TSO + W2W
- NSW = PSO + R2R

Simulation of TSO under PSO:

Add a write-write fence (wfence) before each write
NSW: Non Speculative Writes

- **TSO** = Read-Local-Write-Early + W2R
- **PSO** = TSO + W2W
- **NSW** = PSO + R2R

Simulation of TSO under PSO:

*Add a write-write fence (wfence) before each write*

Simulation of PSO under NSW:

*Add a read-read fence (rfence) before each read*
Operational Model: Event Structures

Process 1:

$p_0\xrightarrow{w(x,1)} p_1 \xrightarrow{w(y,1)} p_2 \xrightarrow{w\text{fence}} p_3 \xrightarrow{r(x,2)} p_4 \xrightarrow{w(y,2)} p_5$

Process 2:

$q_0\xrightarrow{\text{fence}} q_1 \xrightarrow{r(y,1)} q_2 \xrightarrow{w(x,2)} q_3 \xrightarrow{r\text{fence}} q_4 \xrightarrow{r(x,2)} q_5$
Operational Model: Event Structures

Process 1:

\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{wfence} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5 \]

Process 2:

\[ q_0 \xrightarrow{fence} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{rfence} q_4 \xrightarrow{r(x, 2)} q_5 \]

Configuration = control states + memory state + event structures
Operational Model: Event Structures

Process 1:

\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{wfence} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5 \]

Process 2:

\[ q_0 \xrightarrow{fence} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{rfence} q_4 \xrightarrow{r(x, 2)} q_5 \]

Writes on \( x \) are inserted after the last reads, \( wfences \), and writes on \( x \).
Operational Model: Event Structures

Process 1:

\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{\text{wfence}} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5 \]

Process 2:

\[ q_0 \xrightarrow{\text{fence}} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{\text{rfence}} q_4 \xrightarrow{r(x, 2)} q_5 \]

Writes on \( y \) are inserted after the last reads, wfences, and writes on \( y \).

\[ w(y, 1) \]

\[ w(x, 1) \]

\( x = 0 \)

\( y = 0 \)
Operational Model: Event Structures

Process 1:

\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{\text{wfence}} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5 \]

Process 2:

\[ q_0 \xrightarrow{\text{fence}} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{\text{rfence}} q_4 \xrightarrow{r(x, 2)} q_5 \]

Wfences are inserted after the last writes.
Reads on $x$ are inserted after the last writes/reads on $x$. 
Operational Model: Event Structures

Process 1:

\[
\begin{align*}
&\text{p}_0 \xrightarrow{w(x, 1)} \text{p}_1 \xrightarrow{w(y, 1)} \text{p}_2 \xrightarrow{\text{wfence}} \text{p}_3 \xrightarrow{r(x, 2)} \text{p}_4 \xrightarrow{w(y, 2)} \text{p}_5
\end{align*}
\]

Process 2:

\[
\begin{align*}
&\text{q}_0 \xrightarrow{\text{fence}} \text{q}_1 \xrightarrow{r(y, 1)} \text{q}_2 \xrightarrow{w(x, 2)} \text{q}_3 \xrightarrow{\text{rfence}} \text{q}_4 \xrightarrow{r(x, 2)} \text{q}_5
\end{align*}
\]

Writes on \( y \) are inserted after the last reads, wfences, and writes on \( y \).

\[
\begin{align*}
\text{p}_5 &\xrightarrow{w(y, 2)} \text{p}_5 \\
\text{q}_0 &\xrightarrow{r(x, 2)} \text{q}_0
\end{align*}
\]

\[
\begin{align*}
\text{p}_5 &\xrightarrow{w(y, 1)} \text{p}_5 \\
\text{q}_0 &\xrightarrow{r(x, 2)} \text{q}_0
\end{align*}
\]

\[
\begin{align*}
x &= 0 \\
y &= 0
\end{align*}
\]
Operational Model: Event Structures

Process 1: $p_0 \xrightarrow{w(x,1)} p_1 \xrightarrow{w(y,1)} p_2 \xrightarrow{wfence} p_3 \xrightarrow{r(x,2)} p_4 \xrightarrow{w(y,2)} p_5$

Process 2: $q_0 \xrightarrow{fence} q_1 \xrightarrow{r(y,1)} q_2 \xrightarrow{w(x,2)} q_3 \xrightarrow{rfence} q_4 \xrightarrow{r(x,2)} q_5$

Fences are performed by a process only when its event structure is empty.
Operational Model: Event Structures

Process 1: $p_0 \xrightarrow{w(x,1)} p_1 \xrightarrow{w(y,1)} p_2 \xrightarrow{wfence} p_3 \xrightarrow{r(x,2)} p_4 \xrightarrow{w(y,2)} p_5$

Process 2: $q_0 \xrightarrow{fence} q_1 \xrightarrow{r(y,1)} q_2 \xrightarrow{w(x,2)} q_3 \xrightarrow{rfence} q_4 \xrightarrow{r(x,2)} q_5$

Reads on $y$ are inserted after the last writes/reads on $y$. 

$w(y,2)$ $w(y,1)$

$x = 0$

$w(x,1)$ $w(x,2)$

$y = 0$

$r(y,1)$
Operational Model: Event Structures

Process 1:

\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{\text{wfence}} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5 \]

Process 2:

\[ q_0 \xrightarrow{\text{fence}} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{\text{rfence}} q_4 \xrightarrow{r(x, 2)} q_5 \]

**Writes** on \( x \) are inserted after the last **reads**, \( \text{wfences} \), and **writes** on \( x \).
Operational Model: Event Structures

Process 1: \( p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{\text{wfence}} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5 \)

Process 2: \( q_0 \xrightarrow{\text{fence}} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{\text{rfence}} q_4 \xrightarrow{r(x, 2)} q_5 \)

Updates to memory are performed when those writes are minimal.
Operational Model: Event Structures

Process 1:

\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{\text{w fence}} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5 \]

Process 2:

\[ q_0 \xrightarrow{\text{fence}} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{\text{rf fence}} q_4 \xrightarrow{r(x, 2)} q_5 \]

Reads are validated w.r.t. the memory when they are minimal.
Operational Model: Event Structures

Process 1:

\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{\text{w fence}} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5 \]

Process 2:

\[ q_0 \xrightarrow{\text{fence}} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{\text{rfence}} q_4 \xrightarrow{r(x, 2)} q_5 \]

Rfences are performed by a process only if there is no pending reads.
Operational Model: Event Structures

Process 1:
\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{\text{wfence}} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5 \]

Process 2:
\[ q_0 \xrightarrow{\text{fence}} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{\text{rfence}} q_4 \xrightarrow{r(x, 2)} q_5 \]

Reads on \( x \) are validated immediately with the last write on \( x \) (if possible)
Operational Model: Event Structures

Process 1:

\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{wfence} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5 \]

Process 2:

\[ q_0 \xrightarrow{fence} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{rfence} q_4 \xrightarrow{r(x, 2)} q_5 \]

Updates to memory are performed when those writes are minimal.
Operational Model: Event Structures

Process 1:
- $p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{w_{\text{fence}}} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5$

Process 2:
- $q_0 \xrightarrow{\text{fence}} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{r_{\text{fence}}} q_4 \xrightarrow{r(x, 2)} q_5$

Updates to memory are performed when those writes are minimal.

$w(y, 2)$

$p_5$

$q_5$

$x = 2$

$y = 1$
Reads are validated w.r.t. the memory when they are minimal.
Operational Model: Event Structures

Process 1: $p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{wfence} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5$

Process 2: $q_0 \xrightarrow{fence} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{rfence} q_4 \xrightarrow{r(x, 2)} q_5$

Wfences are removed if they are minimal.

$w(y, 2)$

$p_5$

$q_5$

$x = 2$

$y = 1$
Operational Model: Event Structures

Process 1:

\[ p_0 \xrightarrow{w(x, 1)} p_1 \xrightarrow{w(y, 1)} p_2 \xrightarrow{\text{wfence}} p_3 \xrightarrow{r(x, 2)} p_4 \xrightarrow{w(y, 2)} p_5 \]

Process 2:

\[ q_0 \xrightarrow{\text{fence}} q_1 \xrightarrow{r(y, 1)} q_2 \xrightarrow{w(x, 2)} q_3 \xrightarrow{\text{rfence}} q_4 \xrightarrow{r(x, 2)} q_5 \]

Updates to memory are performed when those writes are minimal.
From Event Structures to Buffers

Event Structure Semantics → Event Structure Semantics (without Reads) → Store Buffer Semantics

Read Elimination → Wfence Elimination
From Event Structures to Buffers

Event Structure Semantics → Event Structure Semantics (without Reads) → Store Buffer Semantics

Read Elimination → Wfence Elimination
Elimination of Reads

Configuration = control states + event structures + memory history buffer.

Configuration:
- Control states: p0, q0
- Event structures:
  - w(x, 3) → w(y, 1)
  - w(x, 2) → w(x, 1)
  - w(y, 1) → w(x, 2)

Memory History Buffer:
- x = 2, 2, 2, 1, 0
- y = 1, 1, 0, 0, 0

Events:
- P1: y
- P1: x
- P2: x, y
From Event Structures to Buffers

Event Structure Semantics → Event Structure Semantics (without Reads) → Store Buffer Semantics

Read Elimination → Wfence Elimination
From Event Structures to Buffers

- Event Structure Semantics
- Event Structure Semantics (without Reads)
- Store Buffer Semantics

Read Elimination
Wfence Elimination
Elimination of Write Fences

Configurations = Control states + Variable/Serial Buffers + History Buffer

\[ w(x, 2) \]
\[ w(y, 1) \]
\[ w(x, 1) \]
\[ x = 0 \]
\[ y = 0 \]
\[ P_1, P_2 : x, y \]

Variable Buffers  Serial Buffers  Memory History Buffer
The State Reachability Problem for NSW
Decidability of State Reachability

Approach: Well Structured Systems [Abdulla et al., Finkel et al.]

- Well-Quasi Ordering $\leq$ on Configurations on every sequence $c_0, c_1, c_2, \ldots$, $\exists i < j. \ c_i \leq c_j$
- Monotonicity:
  $\leq$ is a simulation relation w.r.t. transition relation of the model
- $\Rightarrow$ Backward reachability analysis terminates

Problem: NSW ?

- Sub-word ordering on buffers?
  - NSW are Not Monotonic!
- Hard to apply WSS framework to NSW
NSW^+ systems

- NSW ≡ NSW^+
- NSW^+: WSS wrt \preceq

Variable Buffers

```
<table>
<thead>
<tr>
<th>P0</th>
<th>w(x, 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>w(y, 0)</td>
<td></td>
</tr>
</tbody>
</table>
```

```
| P1 : x |
| y = 1 |
| y = 0 |

| P2 : y |
| P1 : x |

```

Single Serial Buffer

```
| x = 1 |
| y = 1 |
```

Memory History Buffer

```
| x = 0 |
| y = 0 |

P1, P2 : x, y
```
NSW\(^+\) systems

- NSW \(\equiv\) NSW\(^+\)
- NSW\(^+\): WSS wrt \(\preceq\)

**NSW\(^+\)** systems

- NSW \(\equiv\) NSW\(^+\)
- NSW\(^+\): WSS wrt \(\preceq\)

```
<table>
<thead>
<tr>
<th>P0</th>
<th>P1, P2 : x, y</th>
</tr>
</thead>
<tbody>
<tr>
<td>w(x, 2)</td>
<td>x = 1</td>
</tr>
<tr>
<td></td>
<td>y = 1</td>
</tr>
<tr>
<td>w(y, 0)</td>
<td>y = 0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Variable Buffers**

**Single Serial Buffer**

**Memory History Buffer**
NSW systems

- NSW $\equiv$ NSW$^+$
- NSW$^+$: WSS wrt $\preceq$

Each message in the serial buffer contains a snapshot of memory

Variable Buffers

Single Serial Buffer

Memory History Buffer

$P_0$

$q_0$

$w(x, 2)$

$w(y, 0)$

$x = 1 \quad x = 1$

$y = 1 \quad y = 0$

$P_2 : y \quad P_1 : x$

$x = 0$

$y = 0$

$P_1, P_2 : x, y$
NSW$^+$ systems

- NSW $\equiv$ NSW$^+$
- NSW$^+$: WSS wrt $\preceq$

Unbounded buffers but lossy

Variable Buffers

Single Serial Buffer

Memory History Buffer

$P_0$

$P_1, P_2 : x, y$

$x = 1$

$y = 1$

$q_0$

$x = 0$

$y = 0$

$P_2 : y$

$P_1 : x$

$w(x, 2)$

$w(y, 0)$
**NSW+ systems**

- NSW $\equiv$ NSW+
- NSW+: WSS wrt $\preceq$

Processes have different views of memory (the use of pointers)

**Single Serial Buffer**

- $P_1$: $x = 1$, $y = 0$
- $P_2$: $w(x, 2)$, $w(y, 0)$

**Variable Buffers**

- $P_0$, $q_0$

**Memory History Buffer**

- $P_1, P_2$: $x, y$
State Reachability: Under approximate analysis

- What is a suitable bounding notion?
- Should allow a compositional reduction to SC
- Should avoid representing the contents of store buffers
K-round Reachability

\[ \text{Run} = P_1 M_{i1} P_{i2} M_{i2} P_{i3} M_{i3} \ldots \]

K-round bounded: \( \forall i. T_i \) has \( \leq K \) rounds
Compositional Reasoning
Encoding Store Buffers: The View of a Process

Mask: \text{Var} \rightarrow \{0, 1\}

Queue: \text{Var} \rightarrow \mathbb{D} \cup \{1\}
Simulating Round 1

[Diagram showing relationships between memory elements and operations]

A. Bouajjani (LIAFA, UP7)
Bounding Store Ages

\[ \begin{array}{c}
i \\
i+1 \\
i+2 \\
i+K \\
1 \\
k \\
\end{array} \]
Translation: $Mask_j$ and $Queue_j$ are used circularly (modulo $K + 1$).
Consequences

- $K$-round reachability is decidable for boolean concurrent programs with recursive procedure calls.

- $K$-store-age reachability is decidable for boolean concurrent programs with finite-state threads (without recursion).

- These results hold also for programs with parametric/dynamic number of threads. (Reduction to coverability in Petri nets, using [Atig, B., Qadeer, 2009] for programs with recursion)

- It is possible to use existing tools for the analysis/verification/testing of concurrent programs under SC.
State Reachability: Conclusion

- State Reachability: Decidable for TSO and beyond. Undecidability when speculative writes are allowed.
- But it is a hard problem (nonprimitive recursive when decidable)!
- However, it is possible to have efficient analysis techniques
- Reduction to SC is a promising idea, can be generalized beyond TSO
- Abstraction-based techniques:
  
  \[e.g., \text{[Kuperstein, Vechev, Yahav, PLDI’11]}\]

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

- Other important models: PowerPC, ARM (hardware), C++