Parameterized Model Checking with Partial Order Reduction Technique for the TSO Weak Memory Model

Sylvain Conchon\textsuperscript{1}, David Declerck\textsuperscript{2}, Fatiha Zaïdi\textsuperscript{1}

\textsuperscript{1}Université Paris-Saclay, \textsuperscript{2}OCamlPro SAS

Cambium Seminar
January 27th, 2020
Initial state: \( x = 0, y = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov [x], 1</code></td>
<td><code>mov [y], 1</code></td>
</tr>
<tr>
<td><code>mov eax, [y]</code></td>
<td><code>mov ebx, [x]</code></td>
</tr>
</tbody>
</table>

**Possible outcomes in \((t1:eax, t2:ebx)\):**
- Obviously: \((0, 1), (1, 0), (1, 1)\)
- Surprisingly: \((0, 0)\)

TSO is a weak memory model: orders of memory accesses \(\neq\) interleaving of instructions.
Initial state: \( x = 0, \ y = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [x], 1</td>
<td>mov [y], 1</td>
</tr>
<tr>
<td>mov eax, [y]</td>
<td>mov ebx, [x]</td>
</tr>
</tbody>
</table>

Possible outcomes in \((t1:eax, t2:ebx)\):
- Obviously: \((0, 1), (1, 0), (1, 1)\)
**Context - part 1: the x86-TSO Memory Model**

<table>
<thead>
<tr>
<th>Initial state: ( x = 0, y = 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>mov [x], 1</td>
</tr>
<tr>
<td>mov eax, [y]</td>
</tr>
<tr>
<td>Thread 2</td>
</tr>
<tr>
<td>mov [y], 1</td>
</tr>
<tr>
<td>mov ebx, [x]</td>
</tr>
</tbody>
</table>

Possible outcomes in \((t1:eax, t2:ebx)\):  
- Obviously: \((0, 1), (1, 0), (1, 1)\)  
- Surprisingly: \((0, 0)\)
Context - part 1: the x86-TSO Memory Model

<table>
<thead>
<tr>
<th>Initial state: x = 0, y = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>mov [x], 1</td>
</tr>
<tr>
<td>mov eax, [y]</td>
</tr>
</tbody>
</table>

Possible outcomes in (t1:eax, t2:ebx):

- Obviously: (0, 1), (1, 0), (1, 1)
- Surprisingly: (0, 0)

TSO is a weak memory model:
orders of memory accesses ≠ interleaving of instructions
Context - **part 1: the x86-TSO memory model**

Eliminating TSO behaviors

New behaviors are not necessarily incorrect

**Memory fences** may be used to prevent some of these behaviors

<table>
<thead>
<tr>
<th>Initial state : $x = 0$, $y = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>mov $[x]$, 1</td>
</tr>
<tr>
<td><strong>mfence</strong></td>
</tr>
<tr>
<td>mov eax, $[y]$</td>
</tr>
</tbody>
</table>
Parameterized systems:
- concurrent systems with an arbitrary number of processes
- expressed as **transition systems** manipulating **arrays** indexed by process identifiers

Example:
- mutual exclusion algorithms
- synchronization barriers
- cache coherence protocols
- ...

Context - part 2: Parameterized Systems
Cubicle is an open source **SMT based model checker**, written in OCaml and its implementation relies on a lightweight and enhanced version of the SMT solver **Alt-Ergo**.

Cubicle implements the **Model Checking Modulo Theories** framework of S. Ghilardi and S. Ranise.

**MCMT = SMT + Backward Reachability Algorithm**
Context - part 3: Running Example

\[
\begin{align*}
\text{type state} & = \text{L1} \mid \text{L2} \mid \text{L3} \\
\text{array} \quad \text{PC}_{\text{proc}} & : \text{state} \\
\text{var} \quad X & : \text{int} \\
\text{var} \quad Y & : \text{int} \\
\text{array} \quad \text{R1}_{\text{proc}} & : \text{int} \\
\text{array} \quad \text{R2}_{\text{proc}} & : \text{int} \\
\text{init}(k) & \{ \text{PC}_k = \text{L1} \land X = 0 \land Y = 0 \land \text{R1}_k \neq 0 \land \text{R2}_k \neq 0 \} \\
\text{unsafe}(i, j) & \{ \text{PC}_i = \text{L3} \land \text{PC}_j = \text{L3} \land \text{R1}_i = 0 \land \text{R2}_j = 0 \} \\
\text{transition } t_{1_1}(k) & \text{requires} \{ \text{PC}_k = \text{L1} \} \quad \{ \text{PC}_k := \text{L2}; X := 1 \} \\
\text{transition } t_{1_2}(k) & \text{requires} \{ \text{PC}_k = \text{L1} \} \quad \{ \text{PC}_k := \text{L2}; Y := 1 \} \\
\text{transition } t_{2_1}(k) & \text{requires} \{ \text{PC}_k = \text{L2} \} \quad \{ \text{PC}_k := \text{L3}; \text{R1}_k := Y \} \\
\text{transition } t_{2_2}(k) & \text{requires} \{ \text{PC}_k = \text{L2} \} \quad \{ \text{PC}_k := \text{L3}; \text{R2}_k := X \} 
\end{align*}
\]
Context - part 3: Running Example

```
type state = L1 | L2 | L3
array PC[proc] : state
var X : int
var Y : int
array R1[proc] : int
array R2[proc] : int
```

![Diagram showing transitions and assignments]

- **Transition t1_1**: Requires \( PC[k] = L1 \) \( \Rightarrow \) \( PC[k] := L2; X := 1 \)
- **Transition t1_2**: Requires \( PC[k] = L1 \) \( \Rightarrow \) \( PC[k] := L2; Y := 1 \)
- **Transition t2_1**: Requires \( PC[k] = L2 \) \( \Rightarrow \) \( PC[k] := L3; R1[k] := Y \)
- **Transition t2_2**: Requires \( PC[k] = L2 \) \( \Rightarrow \) \( PC[k] := L3; R2[k] := X \)
type state = L1 | L2 | L3
array PC[proc] : state
var X : int
var Y : int
array R1[proc] : int
array R2[proc] : int

init (k) { PC[k] = L1
&& X = 0 && Y = 0
&& R1[k]<>0 && R2[k]<0 }
Context - part 3: Running Example

```
type state = L1 | L2 | L3
array PC[proc] : state
var X : int
var Y : int
array R1[proc] : int
array R2[proc] : int

init (k) { PC[k] = L1
    && X = 0 && Y = 0
    && R1[k]<>0 && R2[k]<>0 }

unsafe (i j) {
    PC[i] = L3 && PC[j] = L3 &&
    R1[i] = 0 && R2[j] = 0 }
```
Context - part 3: Running Example

\[
\text{type state} = \text{L1} \mid \text{L2} \mid \text{L3} \\
\text{array PC}[\text{proc}] : \text{state} \\
\text{var X : int} \\
\text{var Y : int} \\
\text{array R1}[\text{proc}] : \text{int} \\
\text{array R2}[\text{proc}] : \text{int}
\]

\[
\text{init} (k) \{ \text{PC}[k] = \text{L1} \\
    \&\& \text{X} = 0 \&\& \text{Y} = 0 \\
    \&\& \text{R1}[k]<>0 \&\& \text{R2}[k]<>0 \}
\]

\[
\text{unsafe} (i \ j) \{ \\
\text{PC}[i] = \text{L3} \&\& \text{PC}[j] = \text{L3} \&\& \\
\text{R1}[i] = 0 \&\& \text{R2}[j] = 0 \}
\]

\[
\text{transition t1_1} (k) \text{ requires} \{ \text{PC}[k] = \text{L1} \} \\
\{ \text{PC}[k] := \text{L2}; \text{X} := 1 \}
\]

\[
\text{transition t1_2} (k) \text{ requires} \{ \text{PC}[k] = \text{L1} \} \\
\{ \text{PC}[k] := \text{L2}; \text{Y} := 1 \}
\]

\[
\text{transition t2_1} (k) \text{ requires} \{ \text{PC}[k] = \text{L2} \} \\
\{ \text{PC}[k] := \text{L3}; \text{R1}[k] := \text{Y} \}
\]

\[
\text{transition t2_2} (k) \text{ requires} \{ \text{PC}[k] = \text{L2} \} \\
\{ \text{PC}[k] := \text{L3}; \text{R2}[k] := \text{X} \}
\]
Context - part 3 : Running Example

```plaintext
(type state = L1 | L2 | L3
array PC[proc] : state
var X : int
var Y : int
array R1[proc] : int
array R2[proc] : int

init (k) { PC[k] = L1
    && X = 0 && Y = 0
    && R1[k]<>0 && R2[k]<>0 }

unsafe (i j) {
    PC[i] = L3 && PC[j] = L3 &&
    R1[i] = 0 && R2[j] = 0 }

transition t1_1 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; X := 1 }

transition t1_2 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; Y := 1 }
```

```plaintext
transition t2_1 (k)
requires { PC[k] = L2 }
{ PC[k] := L3; R1[k] := Y }

transition t2_2 (k)
requires { PC[k] = L2 }
{ PC[k] := L3; R2[k] := X }
```
Context - part 3: Running Example

```
import type state = L1 | L2 | L3
array PC[proc] : state
var X : int
var Y : int
array R1[proc] : int
array R2[proc] : int

init (k) { PC[k] = L1
            && X = 0 && Y = 0
            && R1[k]<0 && R2[k]<0 }

unsafe (i j) { PC[i] = L3 && PC[j] = L3 &&
               R1[i] = 0 && R2[j] = 0 }

transition t1_1 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; X := 1 }

transition t1_2 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; Y := 1 }

transition t2_1 (k)
requires { PC[k] = L2 }
{ PC[k] := L3; R1[k] := Y }

transition t2_2 (k)
requires { PC[k] = L2 }
{ PC[k] := L3; R2[k] := X }
```
Context - part 3: Running Example

```
type state = L1 | L2 | L3
array PC[proc] : state
var X : int
var Y : int
array R1[proc] : int
array R2[proc] : int

init (k) { PC[k] = L1 
  && X = 0 && Y = 0 
  && R1[k]<>0 && R2[k]<>0 }

unsafe (i j) {
  PC[i] = L3 && PC[j] = L3 &&
  R1[i] = 0 && R2[j] = 0 }

transition t1_1 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; X := 1 }

transition t1_2 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; Y := 1 }

transition t2_1 (k)
requires { PC[k] = L2 }
{ PC[k] := L3; R1[k] := Y }

transition t2_2 (k)
requires { PC[k] = L2 }
{ PC[k] := L3; R2[k] := X }
```
If $X$ and $Y$ are weak memories, this algorithm should be considered as unsafe

\[
\text{init}(\#1, \#2) \xrightarrow{t_{1-1}(\#1)} X = 1 \quad \xrightarrow{t_{2-1}(\#1)} R1[\#1] = 0 \quad \xrightarrow{t_{1-2}(\#2)} Y = 1 \quad \xrightarrow{t_{2-2}(\#2)} R2[\#2] = 0
\]
Context - part 3 : Running Example

If $X$ and $Y$ are weak memories, this algorithm should be considered as **unsafe**

\[
\text{init}(#1,#2) \xrightarrow{t_{1-1}(#1)} X = 1 \xrightarrow{t_{2-1}(#1)} R_1[#1] = 0 \xrightarrow{t_{1-2}(#2)} Y = 1 \xrightarrow{t_{2-2}(#2)} R_2[#2] = 0
\]

Unfortunately, the memory model underlying Cubicle is sequential consistency (SC): all reads and writes are in order.
Context - part 3: Running Example

If $X$ and $Y$ are weak memories, this algorithm should be considered as unsafe

\[
\text{init}(\#1, \#2) \xrightarrow{t_{1-1}(\#1)} X = 1 \xrightarrow{t_{2-1}(\#1)} R1[\#1] = 0 \xrightarrow{t_{1-2}(\#2)} X = 1 \xrightarrow{t_{2-2}(\#2)} R2[\#2] = 0
\]

Unfortunately, the memory model underlying Cubicle is sequential consistency (SC): all reads and writes are in order.

Demo
Our goal in this work

Implementing a new version of Cubicle, called Cubicle-$\mathcal{W}$, for the TSO memory model
**Input Language of Cubicle-W**

```plaintext
type state = L1 | L2 | L3
array PC[proc] : state
weak var X : int
weak var Y : int
array R1[proc] : int
array R2[proc] : int

init (k) { PC[k] = L1
           && X = 0 && Y = 0
           && R1[k]<0 && R2[k]<0 }

unsafe (i j) {
    PC[i] = L3 && PC[j] = L3 &&
    R1[i] = 0 && R2[j] = 0 }

transition t1_1 ([k])
requires { PC[k] = L1 }
{ PC[k] := L2; X := 1 }

transition t1_2 ([k])
requires { PC[k] = L1 }
{ PC[k] := L2; Y := 1 }

transition t2_1 ([k])
requires { PC[k] = L2 }
{ PC[k] := L3; R1[k] := Y }

transition t2_2 ([k])
requires { PC[k] = L2 }
{ PC[k] := L3; R2[k] := X }
```
In the rest of the talk

Axiomatic weak memory models
Model checking modulo theories for weak memory models
A TSO-specific partial order reduction technique
Experimental evaluation with Cubicle-$\mathcal{W}$
Conclusion
Axiomatic Weak Memory Models
An axiomatic description of the TSO memory model

TSO reasoning is done through an axiomatic model that:

- maps memory instructions to read and write events
- builds various relations over these events, according to their dependencies

  - po: program order
  - ppo: preserved program order
  - rf: read-from
  - co: coherence
  - fr: from-read
  - fence: memory fence

- builds a *global happens-before (ghb)* relation out of the different relations
An axiomatic description of the TSO memory model

TSO reasoning is done through an axiomatic model that:

- maps memory instructions to read and write events
- builds various relations over these events, according to their dependencies
  - \( po \) : program order
  - \( ppo \) : preserved program order
  - \( rf \) : read-from
  - \( co \) : coherence
  - \( fr \) : from-read
  - \( fence \) : memory fence
- builds a \textit{global happens-before (ghb)} relation out of the different relations

An execution is feasible if we can build an \textit{acyclic ghb} relation
An axiomatic description of the TSO memory model

TSO reasoning is done through an axiomatic model that:

- maps memory instructions to read and write events
- builds various relations over these events, according to their dependencies
  - \textit{po}: program order
  - \textit{ppo}: preserved program order
  - \textit{rf}: read-from
  - \textit{co}: coherence
  - \textit{fr}: from-read
  - \textit{fence}: memory fence
- builds a \textit{global happens-before (ghb)} relation out of the different relations

An execution is \textbf{feasible} if we can build an \textbf{acyclic ghb} relation

We use the formalism of J. Alglave and L. Maranget
Axiomatic TSO model

Events
Memory operations generate events
### Axiomatic TSO model

**Events**

Memory operations generate events

<table>
<thead>
<tr>
<th>Initial state: $x = 0, y = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>mov $[x]$, 1</td>
</tr>
<tr>
<td>mfence</td>
</tr>
<tr>
<td>mov eax, $[y]$</td>
</tr>
</tbody>
</table>
Axiomatic TSO model

Events
Memory operations generate events

<table>
<thead>
<tr>
<th>Initial state: ( x = 0, y = 0 )</th>
<th>( e_1: Wx=0 )</th>
<th>( e_2: Wy=0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Thread 2</td>
<td></td>
</tr>
<tr>
<td>mov ([x], 1)</td>
<td>mov ([y], 1)</td>
<td></td>
</tr>
<tr>
<td>mfence</td>
<td>mfence</td>
<td></td>
</tr>
<tr>
<td>mov eax, ([y])</td>
<td>mov ebx, ([x])</td>
<td></td>
</tr>
</tbody>
</table>

\( e_3: Wx=1 \) \hspace{1cm} \( e_5: Wy=1 \) \hspace{1cm} \( e_4: Ry=? \) \hspace{1cm} \( e_6: Rx=? \)
Axiomatic TSO model

Program Order (po)

Events from the same process are in Program Order

<table>
<thead>
<tr>
<th>Initial state: $x = 0, y = 0$</th>
<th>$e_1: Wx=0$</th>
<th>$e_2: Wy=0$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov [x], 1</td>
<td>mov [y], 1</td>
<td></td>
</tr>
<tr>
<td>mfence</td>
<td>mfence</td>
<td></td>
</tr>
<tr>
<td>mov eax, [y]</td>
<td>mov ebx, [x]</td>
<td></td>
</tr>
<tr>
<td>$e_3: Wx=1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>po</td>
<td></td>
<td>po</td>
</tr>
<tr>
<td>$e_4: Ry=?$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e_5: Wy=1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e_6: Rx=?$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Axiomatic TSO model

Preserved Program Order (ppo)
Under TSO, WR pairs are not preserved in Program Order

Initial state: \( x = 0, \ y = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mov ([x], 1)</td>
<td>mov ([y], 1)</td>
<td></td>
</tr>
<tr>
<td>mfence</td>
<td>mfence</td>
<td></td>
</tr>
<tr>
<td>mov eax, [y]</td>
<td>mov ebx, [x]</td>
<td></td>
</tr>
</tbody>
</table>

\( e_1:Wx=0 \quad e_2:Wy=0 \)

\( e_3:Wx=1 \quad e_5:Wy=1 \)

\( e_4:Ry=? \quad e_6:Rx=? \)
Axiomatic TSO model

Fence
All WR pairs separated by a fence are in a Fence relation

Initial state: \( x = 0, \ y = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{mov} \ [x], 1 )</td>
<td>( \text{mov} \ [y], 1 )</td>
</tr>
<tr>
<td>( \text{mfence} )</td>
<td>( \text{mfence} )</td>
</tr>
<tr>
<td>( \text{mov eax, [y]} )</td>
<td>( \text{mov ebx, [x]} )</td>
</tr>
</tbody>
</table>

\( e_1: Wx=0 \) \( e_2: Wy=0 \)

\( e_3: Wx=1 \) \( e_5: Wy=1 \)

\( e_4: Ry=? \) \( e_6: Rx=? \)
Axiomatic TSO model

Read-From (rf)
Each read takes its value from a single write

| Initial state: \(x = 0, y = 0\) |
|-----------------|-----------------|
| Thread 1        | Thread 2        |
| mov [x], 1      | mov [y], 1      |
| mfence          | mfence          |
| mov eax, [y]    | mov ebx, [x]    |

\(e_1: Wx=0\) \(e_2: Wy=0\)
\(e_3: Wx=1\) \(e_5: Wy=1\)
\(e_4: Ry=0\) \(e_6: Rx=0\)
Axiomatic TSO model

Read-From (rf)
Each read takes its value from a single write

<table>
<thead>
<tr>
<th>Initial state : $x = 0, y = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>mov $[x], 1$</td>
</tr>
<tr>
<td>mfence</td>
</tr>
<tr>
<td>mov eax, $[y]$</td>
</tr>
</tbody>
</table>

| $e_1: Wx=0$ | $e_2: Wy=0$ |
| $e_3: Wx=1$ | $e_5: Wy=1$ |
| $e_4: Ry=1$ | $e_6: Rx=1$ |
Axiomatic TSO model

Read-From (rf)
Each read takes its value from a single write

<table>
<thead>
<tr>
<th>Initial state : $x = 0, y = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>mov $[x]$, 1</td>
</tr>
<tr>
<td>mfence</td>
</tr>
<tr>
<td>mov eax, $[y]$</td>
</tr>
</tbody>
</table>

$e_1: Wx=0$  $e_2: Wy=0$

$e_3: Wx=1$  $e_5: Wy=1$

$e_4: Ry=0$  $e_6: Rx=1$
Axiomatic TSO model

Read-From (rf)
Each read takes its value from a single write

Initial state: \( x = 0, y = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [x], 1</td>
<td>mov [y], 1</td>
</tr>
<tr>
<td>mfence</td>
<td>mfence</td>
</tr>
<tr>
<td>mov eax, [y]</td>
<td>mov ebx, [x]</td>
</tr>
</tbody>
</table>

\[ e_1:Wx=0 \quad e_2:Wy=0 \]

\[ e_3:Wx=1 \quad e_5:Wy=1 \]

\[ e_4:Ry=1 \quad e_6:Rx=0 \]
Axiomatic TSO model

Read-From (rf)
Each read takes its value from a single write

<table>
<thead>
<tr>
<th>Initial state</th>
<th>x = 0, y = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td></td>
</tr>
<tr>
<td>mov [x], 1</td>
<td></td>
</tr>
<tr>
<td>mfence</td>
<td></td>
</tr>
<tr>
<td>mov eax, [y]</td>
<td></td>
</tr>
<tr>
<td>Thread 2</td>
<td></td>
</tr>
<tr>
<td>mov [y], 1</td>
<td></td>
</tr>
<tr>
<td>mfence</td>
<td></td>
</tr>
<tr>
<td>mov ebx, [x]</td>
<td></td>
</tr>
</tbody>
</table>

rf is split in two sub-relations, rfi (internal) and rfe (external), depending on whether it relates to events issued by the same process or events issued by distinct processes.
## Coherence (co)

There is a total order on all writes to the same variable.

### Initial state: $x = 0, y = 0$

<table>
<thead>
<tr>
<th></th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [x], 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mfence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov eax, [y]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $e_1: Wx = 0$
- $e_2: Wy = 0$
- $e_3: Wx = 1$
- $e_4: Ry = ?$
- $e_5: Wy = 1$
- $e_6: Rx = ?$

"When a read takes its value from a write, then a write that is after this specific write in the co relation also has to occur after the read."
Axiomatic TSO model

Coherence (co)
There is a total order on all writes to the same variable

Initial state: $x = 0, y = 0$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov $[x], 1$</td>
<td>mov $[y], 1$</td>
</tr>
<tr>
<td>mfence</td>
<td>mfence</td>
</tr>
<tr>
<td>mov eax, $[y]$</td>
<td>mov ebx, $[x]$</td>
</tr>
</tbody>
</table>

$e_1: Wx = 0 \quad e_2: Wy = 0$

$e_3: Wx = 1 \quad e_5: Wy = 1$

$e_4: Ry = ? \quad e_6: Rx = ?$

From $rf$ and $co$, we derive a new relation $fr$:

$$\forall e_1, e_2, e_3. rf(e_1, e_2) \land co(e_1, e_3) \rightarrow fr(e_2, e_3)$$
Axiomatic TSO model

**Coherence** (co)
There is a total order on all writes to the same variable

<table>
<thead>
<tr>
<th>Initial state : ( x = 0, y = 0 )</th>
<th>( e_1:Wx=0 )</th>
<th>( e_2:Wy=0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>co</td>
<td>( e_3:Wx=1 )</td>
</tr>
<tr>
<td>mov ([x], 1)</td>
<td>( \rightarrow )</td>
<td>e_5:Wy=1</td>
</tr>
<tr>
<td>mfence</td>
<td>( e_4:Ry=? )</td>
<td></td>
</tr>
<tr>
<td>mov eax, ([y])</td>
<td></td>
<td>( e_6:Rx=? )</td>
</tr>
<tr>
<td>Thread 2</td>
<td>co</td>
<td></td>
</tr>
<tr>
<td>mov ([y], 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mfence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mov ebx, ([x])</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From \( rf \) and \( co \), we derive a new relation \( fr \):

\[
\forall e_1, e_2, e_3. rf(e_1, e_2) \land co(e_1, e_3) \rightarrow fr(e_2, e_3)
\]

"When a read takes its value from a write, then a write that is after this specific write in the co relation also has to occur after the read"
Axiomatic TSO model: Global Happens-Before

*ghb* is the *smallest* partial order relation such that:

\[
\forall e_1, e_2 \cdot ppo(e_1, e_2) \implies ghb(e_1, e_2) \quad \text{GHB-PPO}
\]

\[
\forall e_1, e_2 \cdot fence(e_1, e_2) \implies ghb(e_1, e_2) \quad \text{GHB-FENCE}
\]

\[
\forall e_1, e_2 \cdot rfe(e_1, e_2) \implies ghb(e_1, e_2) \quad \text{GHB-RFE}
\]

\[
\forall e_1, e_2 \cdot co(e_1, e_2) \implies ghb(e_1, e_2) \quad \text{GHB-CO}
\]

\[
\forall e_1, e_2 \cdot fr(e_1, e_2) \implies ghb(e_1, e_2) \quad \text{GHB-FR}
\]
Axiomatic TSO model: Global Happens-Before

*ghb* is the **smallest** partial order relation such that:

\[
\forall e_1, e_2 \cdot ppo(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-PPO}
\]

\[
\forall e_1, e_2 \cdot fence(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-FENCE}
\]

\[
\forall e_1, e_2 \cdot rfe(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-RFE}
\]

\[
\forall e_1, e_2 \cdot co(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-CO}
\]

\[
\forall e_1, e_2 \cdot fr(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-FR}
\]

An execution defined by \((po, rf, co, fence)\) is **feasible** if the *ghb* relation that it generates is **acyclic**
Without fences, an execution that ends with \((eax=0, ebx=0)\) is feasible.

**Initial state:** \(x = 0, y = 0\)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov [x], 1</td>
<td>mov [y], 1</td>
</tr>
<tr>
<td>mov eax, [y]</td>
<td>mov ebx, [x]</td>
</tr>
</tbody>
</table>

**Events:**
- \(e_1: Wx=0\)
- \(e_2: Wy=0\)
- \(e_3: Wx=1\)
- \(e_4: Ry=0\)
- \(e_5: Wy=1\)
- \(e_6: Rx=0\)
Axiomatic TSO Model: invalid execution

Using memory barriers, such execution is not feasible

<table>
<thead>
<tr>
<th>Etat initial: ( x = 0, \ y = 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread 1</strong></td>
</tr>
<tr>
<td>mov ([x], 1)</td>
</tr>
<tr>
<td>mfence</td>
</tr>
<tr>
<td>mov eax, ([y])</td>
</tr>
</tbody>
</table>

\( e_1: Wx = 0 \) \quad e_2: Wy = 0 

\( e_3: Wx = 1 \) \quad e_4: Ry = 0 

\( e_5: Wy = 1 \) \quad e_6: Rx = 0 

\( \text{co} \) \quad \text{rf} 

\( \text{fr} \) \quad \text{fr} 

\( \text{fence} \)
MCMT for Weak Memory Models
MCMT [Ghilardi, Ranise]

System states and transitions are first-order formulas
MCMT [Ghilardi, Ranise]

System states and transitions are first-order formulas

**Initial states** are defined by a *universally* quantified formula:

\[
\text{init}(i) \{ \ A[i] = \text{True} \land PC = L1 \ }
\]

\[
\forall i : \text{proc}.A[i] \land PC = L1
\]

**Bad states** are defined by special *existentially* quantified formulas, called **cubes**:

\[
\text{unsafe}(i \ j) \{ \ S[i] = \text{Crit} \land S[j] = \text{Crit} \ }
\]

\[
\exists i, j : \text{proc}.i \neq j \land S[i] = \text{Crit} \land S[j] = \text{Crit}
\]

**Transitions** correspond to *existentially* quantified formulas:

\[
\text{transition } t(i) \text{ requires } \{ \ S[i] = A \land PC = L1 \ }
\]

\[
\{ \ S[i] = B ; X = X+1 \ }
\]

\[
\exists i : \text{proc}.S[i] = A \land PC = L1 \land S' = S[i <- B] \land X' = X + 1
\]
Inductive invariants

We are looking for a predicate $\text{Reach}$ such that:

Reach is an inductive invariant:

$$\forall \vec{x}. \text{Init}(\vec{x}) \Rightarrow \text{Reach}(\vec{x})$$

$$\forall \vec{x}, \vec{x}'. \text{Reach}(\vec{x}) \land \tau(\vec{x}, \vec{x}') \Rightarrow \text{Reach}(\vec{x}')$$

The system is safe if there exists an interpretation of Reach such that:

$$\forall \vec{x}. \text{Reach}(\vec{x}) \models \neg \text{unsafe}(\vec{x})$$
Inductive invariants

We are looking for a predicate \textit{Reach} such that :

Reach is an \textit{inductive invariant} :

\[
\forall \vec{x}. \text{Init}(\vec{x}) \Rightarrow \text{Reach}(\vec{x})
\]

\[
\forall \vec{x}, \vec{x}'. \text{Reach}(\vec{x}) \land \tau(\vec{x}, \vec{x}') \Rightarrow \text{Reach}(\vec{x}')
\]

The system is \textit{safe} if there exists an interpretation of Reach such that :

\[
\forall \vec{x}. \text{Reach}(\vec{x}) \models \neg \text{unsafe}(\vec{x})
\]

Cubicle computes \textit{Reach} by \textit{backward reachability}
Backward Reachability

\[ \text{BR} (\tau, I, U): \]
\[ V := \emptyset \]
\[ \text{push}(Q, U) \]
\[ \text{while} \ \text{not}_\text{empty}(Q) \ \text{do} \]
\[ \varphi := \text{pop}(Q) \]
\[ \text{if} \ \varphi \land I \ \text{sat} \ \text{then} \]
\[ \text{return} \ \text{unsafe} \]
\[ \text{if} \ \neg (\varphi \models \bigvee_{\psi \in V} \psi) \ \text{then} \]
\[ V := V \cup \{ \varphi \} \]
\[ \text{push}(Q, \text{pre}_\tau(\varphi)) \]
\[ \text{return} \ \text{safe} \]
Backward Reachability

\[ \text{BR} (\tau, I, U) : \]
\[
V := \emptyset \\
push(Q, U) \\
\textbf{while} \ not\_empty(Q) \ \textbf{do} \\
\quad \varphi := \text{pop}(Q) \\
\quad \textbf{if} \ \varphi \wedge I \ \text{sat} \ \textbf{then} \\
\quad \quad \text{return unsafe} \\
\quad \textbf{if} \ \neg(\varphi \models \bigvee_{\psi \in V} \psi) \ \textbf{then} \\
\quad \quad \quad V := V \cup \{ \varphi \} \\
\quad \quad \quad \text{push}(Q, \text{pre}_\tau(\varphi)) \\
\quad \text{return safe} \\
\]
Backward Reachability

\[ \text{BR} (\tau, I, U): \]
\[
V := \emptyset \\
\text{push}(Q, U) \\
\text{while not_empty}(Q) \text{ do} \\
\quad \varphi := \text{pop}(Q) \\
\quad \text{if } \varphi \land I \text{ sat then} \\
\quad \quad \text{return unsafe} \\
\quad \text{if } \neg(\varphi \models \bigvee_{\psi \in V} \psi) \text{ then} \\
\quad \quad V := V \cup \{ \varphi \} \\
\quad \quad \text{push}(Q, \text{pre}_\tau(\varphi)) \\
\text{return safe}
\]
Backward Reachability

BR (τ, I, U):
V := ∅
push(Q, U)
while not_empty(Q) do
    ϕ := pop(Q)
    if ϕ ∧ I sat then
        return unsafe
    if ¬(ϕ ⊨ \bigvee_{ψ \in V} ψ) then
        V := V ∪ {ϕ}
push(Q, pre_τ(ϕ))
return safe
Backward Reachability

**BR \((\tau, I, U)\):**

\[
V := \emptyset \\
push(Q, U) \\
\textbf{while} \not_{\text{empty}}(Q) \textbf{ do} \\
\textcolor{red}{\varphi} := \text{pop}(Q) \\
\textbf{if} \; \varphi \land I \text{ sat} \; \textbf{then} \\
\textbf{return} \; \text{unsafe} \\
\textbf{if} \; \neg (\varphi \models \bigvee_{\psi \in V} \psi) \; \textbf{then} \\
V := V \cup \{ \varphi \} \\
push(Q, \text{\textcolor{red}{pre}}_{\tau}(\varphi)) \\
\textbf{return} \; \text{safe}
\]
Backward Reachability

\( \text{BR} (\tau, I, U): \)

\[
V := \emptyset \\
push(Q, U) \\
\text{while not_empty(Q) do} \\
\quad \varphi := \text{pop}(Q) \\
\quad \text{if } \varphi \land I \text{ sat then} \\
\quad \quad \text{return unsafe} \\
\quad \text{if } \neg (\varphi \models \bigvee_{\psi \in V} \psi) \text{ then} \\
\quad \quad V := V \cup \{ \varphi \} \\
\quad \quad \text{push}(Q, \text{pre}_\tau(\varphi)) \\
\quad \text{return safe}
\]
Backward Reachability

\[ \text{BR} \left( \tau, I, U \right): \]

\[
\begin{align*}
V & := \emptyset \\
push(Q, U) \\
\text{while not_empty}(Q) \text{ do} \\
\quad \varphi & := \text{pop}(Q) \\
\quad \text{if } \varphi \land I \text{ sat} \text{ then} \\
\quad \quad & \text{return unsafe} \\
\quad \text{if } \neg (\varphi \models \bigvee_{\psi \in V} \psi) \text{ then} \\
\quad \quad \quad & V := V \cup \{ \varphi \} \\
\quad \quad \quad & \text{push}(Q, \text{pre}_\tau(\varphi)) \\
\text{return} & \text{ safe}
\end{align*}
\]
Backward Reachability

\[ \text{BR} \left( \tau, I, U \right): \]
\[
V := \emptyset \\
push(Q, U) \\
\text{while} \ not\_empty(Q) \ \text{do} \\
\quad \varphi := \text{pop}(Q) \\
\quad \text{if } \varphi \land I \text{ sat then} \\
\quad \quad \text{return unsafe} \\
\quad \text{if } \neg (\varphi \models \bigvee_{\psi \in V} \psi) \ \text{then} \\
\quad \quad \quad V := V \cup \{ \varphi \} \\
\quad \quad \quad \text{push}(Q, \text{pre}_\tau(\varphi)) \\
\quad \quad \text{return safe}
\]
Cubicle-$\mathcal{W}$ implements an extended version of the backward reachability algorithms of Cubicle

For reasoning about TSO, one needs to:

- associate **events** to read and write operations in logical formulas
- build a global happens-before (ghb) relation during the backward analysis according to the dependencies between those events
- add an **axiomatic model** of TSO inside the SMT solver to check satisfiability of TSO formulas
Backward Reachability Modulo TSO

\[ \text{BR} (\tau, I, U): \]
\[
V := \emptyset
\]
\[
push(Q, U)
\]
\[
\text{while not_empty}(Q) \text{ do}
\]
\[
\varphi := \text{pop}(Q)
\]
\[
\text{if } \varphi \land I \text{ sat then}
\]
\[
\text{return unsafe}
\]
\[
\text{if } \neg(\varphi \models \bigvee_{\psi \in V} \psi) \text{ then}
\]
\[
V := V \cup \{ \varphi \}
\]
\[
push(Q, \text{pre}_\tau(\varphi))
\]
\[
\text{return safe}
\]
Backward Reachability Modulo TSO

\( \text{BR} (\tau, I, U): \)

\[
V := \emptyset \\
\text{push}(Q, U) \\
\text{while not_empty}(Q) \text{ do} \\
\quad \varphi := \text{pop}(Q) \\
\quad \text{if } \varphi \land I \text{ sat then} \\
\quad \quad \text{return unsafe} \\
\quad \text{if } \neg(\varphi \models \bigvee_{\psi \in V} \psi) \text{ then} \\
\quad \quad \quad V := V \cup \{ \varphi \} \\
\quad \quad \quad \text{push}(Q, \text{pre}_\tau(\varphi)) \\
\quad \text{return safe}
\]

Safety test & Fixpoint check

- Performed by an SMT solver
- Logic and SMT extended to reason about events and TSO relations

Pre-image computation

- Instrumented to produce events and relations
- Decides which writes satisfy each read
Events

weak var X : int
weak array T[proc] : int
Events

weak var $X : \texttt{int}$
weak array $T[\texttt{proc}] : \texttt{int}$

When a variable is read:
transition $t([i])$ requires \{ $X = 42$ \} \{ $\ldots$ \}

$$\exists e_1. e_1 : R_X^i \land \text{val}(e_1) = 42 \land \text{pending}_X(e_1)$$
Events

weak var X : int
weak array T[proc] : int

When a variable is **read**:
transition \( t([i]) \) requires \{ X = 42 \} \{ ... \}

\[ \exists e_1. e_1: R^i_X \land val(e_1) = 42 \land pending_X(e_1) \]

When a variable is **assigned**:
transition \( t([i]) \) requires \{ ... \} \{ X := 42 \}

\[ \exists e_2. e_2: W^i_X \land val(e_2) = 42 \]
Initial states

type state = L1 | L2 | L3
array PC[proc] : state
weak var X : int
weak var Y : int
array R1[proc] : int
array R2[proc] : int

init (k) {PC[k] = L1 && X = 0 && Y = 0 && R1[k]<>0 && R2[k]<>0 }
Initial states

deprecated

```plaintext
type state = L1 | L2 | L3
array PC[proc] : state
weak var X : int
weak var Y : int
array R1[proc] : int
array R2[proc] : int

init (k) {PC[k] = L1 && X = 0 && Y = 0 && R1[k]<>0 && R2[k]<>0 }

∀k.∀e₁, e₂. PC[k] = L1 ∧ R₁[k] ≠ 0 ∧ R₂[k] ≠ 0 ∧
    e₁ : Rd₁^k X ∧ pending_X(e₁) ∧ val_X(e₁) = 0 ∧
    e₂ : Rd₁^k Y ∧ pending_Y(e₂) ∧ val_Y(e₂) = 0
```
Pre-image Computation

Similar to Cubicle, but …

- connect reads and writes with the \( rf \) relation

\[
e_2: \text{Wr}^{i2}_X, \; \text{val}_X(e_2) = 1 \\
\text{rf}(e_2, e_1), \; \text{val}(e_2) = \text{val}(e_1) \\
e_1: \text{Rd}^{i1}_X, \; \text{val}_X(e_1) > 0, \neg \text{pending}_X(e_1)
\]

\[
e_1: \text{Rd}^{i1}_X, \; \text{val}_X(e_1) > 0, \text{pending}_X(e_1)
\]

- decides the memory coherence between writes with the \( co \) relation

\[
e_2: \text{Wr}^{i2}_X, \; \text{val}(e_2) = 1 \\
e_1: \text{Rd}^{i1}_X, \; \text{val}_X(e_1) > 0, \text{pending}_X(e_1)
\]

\[
e_1: \text{Rd}^{i1}_X, \; \text{val}_X(e_1) > 0, \text{pending}_X(e_1)
\]

\[
e_2: \text{Wr}^{i2}_X, \; \text{val}(e_2) = 1 \\
e_1: \text{Rd}^{i1}_X, \; \text{val}_X(e_1) > 0, \text{pending}_X(e_1)
\]

\[
e_1: \text{Rd}^{i1}_X, \; \text{val}_X(e_1) > 0, \text{pending}_X(e_1)
\]
Running Example

```
type state = L1 | L2 | L3
array PC[proc] : state
weak var X : int
weak var Y : int
array R1[proc] : int
array R2[proc] : int

init (k) { PC[k] = L1
            && X = 0 && Y = 0
            && R1[k]<>0 && R2[k]<>0 }

unsafe (i j) {
    PC[i] = L3 && PC[j] = L3 &&
    R1[i] = 0 && R2[j] = 0 }

transition t1_1 ([k])
  requires { PC[k] = L1 }
  { PC[k] := L2; X := 1 }

transition t1_2 ([k])
  requires { PC[k] = L1 }
  { PC[k] := L2; Y := 1 }

transition t2_1 ([k])
  requires { PC[k] = L2 }
  { PC[k] := L3; R1[k] := Y }

transition t2_2 ([k])
  requires { PC[k] = L2 }
  { PC[k] := L3; R2[k] := X }
```
Backward Reachability Modulo TSO: Example

\[ PC[i_1] = L3 \quad PC[i_2] = L3 \]
\[ R1[i_1] = 0 \quad R2[i_2] = 0 \]
Backward Reachability Modulo TSO: Example

\[ PC[i_1] = \text{L3} \quad PC[i_2] = \text{L3} \]
\[ R1[i_1] = 0 \quad R2[i_2] = 0 \]

\[ e_1 : \text{Rd}^{i_2}_x \quad \text{val}_x(e_1) = 0 \]
\[ \text{pending}_x(e_1) \]
Backward Reachability Modulo TSO : Example

PC[i1] = L2, PC[i2] = L2
\( e_1 : Rd^{i2}_x \) \( val_x(e_1) = 0 \)
\( e_2 : Rd^{i2}_y \) \( val_y(e_2) = 0 \)
\( pending_x(e_1), pending_y(e_2) \)

t1_2(i1)

PC[i1] = L3, PC[i2] = L3
R1[i1] = 0, R2[i2] = 0
\( e_1 : Rd^{i2}_x \) \( val_x(e_1) = 0 \)
\( pending_x(e_1) \)

t2_2(i2)
Backward Reachability Modulo TSO : Example

PC\[i_1\]=L3  PC\[i_2\]=L3
R1[i_1]=0  R2[i_2]=0

PC\[i_1\]=L1  PC\[i_2\]=L1
val_y(e3)=val_y(e2)
e_1:Rd^\{X\}  val_X(e_1) = 0
e_2:Rd^\{Y\}  val_Y(e_2) = 0
e_3:Wri^\{Y\} \land val_Y(e_3) = 1

pending_X(e_1) \neg pending_Y(e_2)
val_Y(e_3)=val_Y(e_2)
po(e_3,e_1)  rf(e_3,e_2)
Backward Reachability Modulo TSO: Example

PC\[i_1\]=L2 PC\[i_2\]=L1
val_\(x\)(e3)=val_\(x\)(e2)
e_1:Rd_\(x\) val_\(x\)(e_1) = 0
e_2:Rd_\(y\) val_\(y\)(e_2) = 0
e_3:Wr_\(y\) \land val_\(y\)(e3) = 1
pending_\(x\)(e_1) pending_\(y\)(e_2)
po(e3,e1)

PC\[i_1\]=L2 PC\[i_2\]=L2
\(e_1:Rd_\(x\)\) val_\(x\)(e_1) = 0
\(e_2:Rd_\(y\)\) val_\(y\)(e_2) = 0
pending_\(x\)(e_1) pending_\(y\)(e_2)

PC\[i_1\]=L3 PC\[i_2\]=L3
R1\[i_1\]=0 R2\[i_2\]=0
\(e_1:Rd_\(x\)\) val_\(x\)(e_1) = 0
pending_\(x\)(e_1)
Backward Reachability Modulo TSO: Example

PC\[i_1\]=L2 PC\[i_2\]=L1
val_x(e_3)=val_y(e_2)
e_1:Rd^2_x val_x(e_1) = 0
e_2:Rd^2_y val_y(e_2) = 0
e_3:Wr^2_y \land val_y(e_3) = 1
pending_x(e_1) pending_y(e_2)
po(e_3,e_1)

PC\[i_1\]=L2 PC\[i_2\]=L1
val_y(e_3)=val_y(e_2)
e_1:Rd^2_x val_x(e_1) = 0
e_2:Rd^2_y val_y(e_2) = 0
e_3:Wr^2_y \land val_y(e_3) = 1
pending_x(e_1) pending_y(e_2)
po(e_3,e_1) rf(e_3,e_2)
Backward Reachability Modulo TSO: Example

$PC[i_1] = L_1 \quad PC[i_2] = L_1$
$val_y(e_3) = val_y(e_2)$
$e_1: Rd_{X}^{i_2} \quad val_X(e_1) = 0$
$e_2: Rd_{Y}^{i_2} \quad val_Y(e_2) = 0$
$e_3: Wr_{X}^{i_2} \land val_Y(e_3) = 1$
$e_4: Wr_{X}^{i_2} \land val_X(e_4) = 1$
$pending_X(e_1) \quad pending_Y(e_2)$
$po(e_3, e_1) \quad po(e_4, e_2)$

$PC[i_1] = L_2 \quad PC[i_2] = L_2$
$e_1: Rd_{X}^{i_2} \quad val_X(e_1) = 0$
$e_2: Rd_{Y}^{i_2} \quad val_Y(e_2) = 0$
$e_3: Wr_{Y}^{i_2} \land val_Y(e_3) = 1$
$pending_X(e_1) \quad pending_Y(e_2)$
$po(e_3, e_1)$

$PC[i_1] = L_2 \quad PC[i_2] = L_1$
$val_y(e_3) = val_y(e_2)$
$e_1: Rd_{X}^{i_2} \quad val_X(e_1) = 0$
$e_2: Rd_{Y}^{i_2} \quad val_Y(e_2) = 0$
$e_3: Wr_{Y}^{i_2} \land val_Y(e_3) = 1$
$pending_X(e_1) \quad pending_Y(e_2)$
$po(e_3, e_1)$
$rf(e_3, e_2)$

$PC[i_1] = L_3 \quad PC[i_2] = L_3$
$R1[i_1] = 0 \quad R2[i_2] = 0$
$e_1: Rd_{X}^{i_2} \quad val_X(e_1) = 0$
$pending_X(e_1) \quad pending_X(e_2)$
A TSO-Specific Partial Order Reduction Technique
A TSO-Specific Partial Order Reduction Technique

Keep only coherence constraints compatible with the scheduling

scheduling(e1, e2)

co(e1, e2)

e1: Wx = 1

e2: Wx = 2

co(e2, e1)

e1: Wx = 1

e2: Wx = 2
Theorem. For every execution defined by \((po, rf, co)\) and every scheduling \(S\) of a program, there exists a scheduling \(S'\) such that \(co\) is compatible with \(S'\).
Efficient Backward Reachability Modulo TSO

We exploit the partial order reduction property to simplify backward search of TSO by directly building the $ghb$ relation, on the fly.

- A new read will be before any old event event from the same process in $ghb$.
- A new write will be before any old write from the same process in $ghb$.
- A new write will be before any old read from the same process in $ghb$ if they are separated by a fence.
- A new write will be before any old write on the same variable in $ghb$ (compatibility of co/sched).
- A new write will be before any old read from a different process that it satisfies in $ghb$.
- A new write will be after any old read from a different process that it does not satisfy in $ghb$.
- A new read will be before any old write on the same variable in $ghb$. 
Benchmarks

Cubicle-$\mathcal{W}$ has been evaluated for several kinds of algorithms:

- **Mutual exclusion**
  - High level: naive mutex, arbitrer, Dekker, Peterson, Burns
  - Assembly code: Spinlock Linux, Mutex/xchg, Mutex/cmpxchg

- **Sense-Reversing Barrier**

- **Two-Phase Commit**
Benchmarks : Other Verification Tools for Weak Memory

Parameterized systems :
Dual-TSO (Abdulla, Atig, Bouajjani, Ngo, Univ. Uppsala & Univ. Paris 7)
→ safety properties

Fix number of processes :
MEMORAX (Abdulla, Atig et al, Univ. Uppsala)
→ safety properties

Trencher (Bouajjani, Calin et al, Univ. Paris 7 & Univ. Kaiserslautern)
→ robustness
→ bug finding

CBMC (Alglave, Kroening et al, Univ. College London & Univ. Oxford)
→ bug finding
→ C code analysis
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cubicle W</th>
<th>Dual TSO</th>
<th>Memorax PB</th>
<th>Trencher</th>
<th>CBMC Unwind 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive mutex</td>
<td>0.30s [N]</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>35.7s [4]</td>
<td>2m27 [10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54.8s [5]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2m24 [4]</td>
</tr>
<tr>
<td>lamport</td>
<td>0.60s [N]</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9.42s [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.37s [4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8m39 [3]</td>
</tr>
<tr>
<td>spinlock</td>
<td>0.06s [N]</td>
<td>TO [N]</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TO [6]</td>
<td></td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1m16 [5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>19.58s [2]</td>
</tr>
<tr>
<td>sense_rev</td>
<td>0.06s [N]</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TO [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.09s [2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12m25 [8]</td>
</tr>
<tr>
<td>arbiter_v2</td>
<td>13.5s [N]</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TO [1+3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24.2s [1+2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.62s [1+3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2m56 [1+3]</td>
</tr>
<tr>
<td>two_phase</td>
<td>54.1s [N]</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TO [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12.3s [2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12m39 [10]</td>
</tr>
</tbody>
</table>

\* = crash of the tool  \* = incorrect answer  TO > 20 minutes
Conclusion and Perspectives

Contributions:
- A Model Checker for TSO parameterized systems
- A partial order reduction technique for TSO
- An extension of Cubicle for TSO called Cubicle-$\mathcal{W}$

Perspectives:
- Generation of invariants
- Other memory models
Thank you

Some papers (by S. Conchon, D. Declerck, F. Zaïdi)

Parameterized model checking with partial order reduction technique for the TSO weak memory model [JAR 2020, to appear]

Cubicle-$\mathcal{W}$: Parameterized Model Checking on Weak Memory [IJCAR 2018]

Compiling Parameterized x86-TSO Concurrent Programs to Cubicle-$\mathcal{W}$ [ICFEM 2017]

Parameterized Model Checking Modulo Explicit Weak Memory Models [IMPEX 2017]

Cubicle-$\mathcal{W}$: http://cubicle.lri.fr/cubiclew/