Compilation of OCaml memory model to Power

Egor Namakonov, Anton Podkopaev
An execution result is explained by alternating threads

<table>
<thead>
<tr>
<th></th>
<th>[x] := 1</th>
<th>a := [x]</th>
<th>b := [y]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[y] :=</td>
<td>[y] := 1</td>
<td>c := [x]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
[x] = [y] = 0
\]
An execution result is explained by alternating threads

\[
\begin{array}{|c|c|c|}
\hline
[x] = [y] = 0 \\
\hline
[x] := 1 & a := [x] & b := [y] \\
\hline
[y] := 1 & c := [x] \\
\hline
\end{array}
\]

\[ a = b = c = 1 ? \]
An execution result is explained by alternating threads

<table>
<thead>
<tr>
<th>[x] = [y] = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x] := 1</td>
</tr>
<tr>
<td>[y] := 1</td>
</tr>
</tbody>
</table>

a = b = c = 1 ?
An execution result is explained by alternating threads

\[
\begin{array}{|c|c|c|}
\hline
& [x] = [y] = 0 & \\
\hline
[x] := 1 & a := [x] & b := [y] \\
\hline
[y] := 1 & c := [x] & \\
\hline
\hline
a = b = c = 1 ? & \\
\hline
\end{array}
\]
An execution result is explained by alternating threads

| [x] := 1 | a := [x] | b := [y] |
| [y] := 1 | c := [x] |

a = b = c = 1?
An execution result is explained by alternating threads

<table>
<thead>
<tr>
<th>[x] = [y] = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x] := 1</td>
</tr>
<tr>
<td>[y] := 1</td>
</tr>
<tr>
<td>a = b = c = 1</td>
</tr>
</tbody>
</table>
An execution result is explained by alternating threads

<table>
<thead>
<tr>
<th>[x] = [y] = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x] := 1</td>
</tr>
<tr>
<td>[y] := 1</td>
</tr>
<tr>
<td>a = b = c = 1?</td>
</tr>
</tbody>
</table>
An execution result is explained by alternating threads…

\[
\begin{align*}
[x] &= [y] = 0 \\
[x] := 1 &\quad a := [x] &\quad b := [y] \\
[y] := 1 &\quad c := [x] \\
a = b = c &= 1
\end{align*}
\]
An execution result is explained by alternating threads

\[
\begin{array}{|c|c|c|}
\hline
[x] & [y] & 0 \\
\hline
[x] := 1 & a := [x] & b := [y] \\
\hline
[y] := 1 & c := [x] \\
\hline
a = b = 1, c = 0 ? \\
\hline
\end{array}
\]
An execution result is explained by alternating threads

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[x] := 1</td>
<td>a := [x]</td>
<td>b := [y]</td>
</tr>
<tr>
<td>[y] := 1</td>
<td>c := [x]</td>
<td></td>
</tr>
</tbody>
</table>

a = b = 1, c = 0 ?
An execution result is explained by alternating threads

\[
\begin{align*}
[x] &= [y] = 0 \\
[x] &:= 1 \quad a := [x] \quad b := [y] \\
[y] &:= 1 \quad c := [x] \\
a &= b = 1, \ c = 0 \ ?
\end{align*}
\]
An execution result is explained by alternating threads

\[
\begin{array}{c|c|c}
   [x] = [y] = 0 \\
   [x] := 1 & a := [x] & b := [y] \\
   [y] := 1 & c := [x] \\
   a = b = 1, c = 0 \\
\end{array}
\]
An execution result is explained by alternating threads

\[
\begin{array}{|c|c|}
\hline
[x] = [y] = 0 \\
\hline
[x] := 1 & a := [x] & b := [y] \\
\hline
[y] := 1 & c := [x] \\
\hline
\end{array}
\]

\[a = b = 1, c = 0 ?\]
An execution result is explained by alternating threads

<table>
<thead>
<tr>
<th></th>
<th>a := [x]</th>
<th>b := [y]</th>
<th>[x] := 1</th>
<th>[y] := 1</th>
<th>c := [x]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x] = [y] = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a = b = 1, c = 0?
An execution result is explained by alternating threads... usually

<table>
<thead>
<tr>
<th>[x] = [y] = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x] := 1</td>
</tr>
<tr>
<td>a := [x]</td>
</tr>
<tr>
<td>b := [y]</td>
</tr>
<tr>
<td>[y] := 1</td>
</tr>
<tr>
<td>c := [x]</td>
</tr>
</tbody>
</table>

a = b = 1, c = 0
C++ allows it due to a (non-atomic) data race

<table>
<thead>
<tr>
<th>[x] = [y] = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x] := 1</td>
</tr>
<tr>
<td>[y] := 1</td>
</tr>
</tbody>
</table>

\[a = b = 1, c = 0\]
C++ allows it due to a (non-atomic) data race

\[
\begin{array}{cccc}
[x] &=& [y] &=& 0 \\
[x] := 1 & a := [x] & b := [y] \\
[y] := 1 & c := [x] \\
a &= b &= 1, c &= 0 \\
\end{array}
\]
C++ allows it due to a (non-atomic) data race

| [x] := 1 | a := [x] | b := [y] |
| [y] := 1 | c := [x] |

a = b = 1, c = 0
C++ allows it due to a (non-atomic) data race

Non-atomic accesses

<table>
<thead>
<tr>
<th></th>
<th>[x] := 1</th>
<th>a := [x]</th>
<th>b := [y]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[y] := 1</td>
<td></td>
<td></td>
<td>c := [x]</td>
</tr>
</tbody>
</table>

a = b = 1, c = 0

Standard for Programming Language C++, 6.8.2.1.20: “Any such data race results in undefined behavior.”
No races on atomics

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

\[
\begin{array}{|c|c|c|}
\hline
[x]_{rlx} & := & 1 \\
\hline
a & := & [x]_{rlx} \\
\hline
b & := & [y]_{rlx} \\
\hline
[y]_{rlx} & := & 1 \\
\hline
c & := & [x]_{rlx} \\
\hline
\end{array}
\]

\[ a = b = 1, c = 0 \]
No races on atomics

$[x] = [y] = 0$

$[x]_{rlx} := 1 \quad a := [x]_{rlx} \quad b := [y]_{rlx}$

$[y]_{rlx} := 1 \quad c := [x]_{rlx}$

$a = b = 1, c = 0$ ?
No races on atomics but the outcome is still allowed

<table>
<thead>
<tr>
<th>[x] = [y] = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x]_{rlx} := 1</td>
</tr>
<tr>
<td>[y]_{rlx} := 1</td>
</tr>
</tbody>
</table>

a = b = 1, c = 0
C++ memory model

\[
\begin{array}{c|c|c}
[x] &= [y] &= 0 \\
[x]_{rlx} := 1 & a := [x]_{rlx} & b := [y]_{rlx} \\
[y]_{rlx} := 1 & c := [x]_{rlx} \\
\end{array}
\]
C++ memory model

\[
\begin{align*}
[x] &= [y] = 0 \\
[x]^{rlx} &:= 1 & a &= [x]^{rlx} & b &= [y]^{rlx} \\
[y]^{rlx} &:= 1 & c &= [x]^{rlx}
\end{align*}
\]

= \{ ..., (a=b=c=1), ..., (a=b=1, c=0), ... \}
C++ memory model is weak

\[
\begin{array}{|c|c|c|}
\hline
[x] = [y] = 0 \\
[x]_{rlx} := 1 & a := [x]_{rlx} & b := [y]_{rlx} \\
[y]_{rlx} := 1 & c := [x]_{rlx} \\
\hline
\end{array}
\]

\[
= \{ \ldots, (a=b=c=1), \ldots \} \]
C++ memory model is **weak** as it allows optimizations

\[
\begin{array}{|c|c|}
\hline
[x] = [y] = 0 \\
\hline
[x]_{rlx} := 1 & a := [x]_{rlx} \\
\hline
[y]_{rlx} := 1 & b := [y]_{rlx} \\
\hline
[y]_{rlx} := 1 & c := [x]_{rlx} \\
\hline
\end{array}
\]

\[
= \{ \ldots, (a=b=c=1), \ldots (a=b=1, c=0), \ldots \}
\]
C++ memory model is **weak** as it allows optimizations.

\[
\begin{array}{|c|c|c|}
\hline
[x] &= [y] &= 0 \\
[x]_{rlx} := 1 & a := [x]_{rlx} & b := [y]_{rlx} \\
[y]_{rlx} := 1 & c := [x]_{rlx} \\
\hline
\end{array}
\]

= \{ ..., (a=b=c=1), ... (a=b=1, c=0), ... \}
C++ memory model is **weak** as it allows optimizations

\[ \begin{array}{|c|c|c|}
\hline
[ x ] \rlr := 1 & a := [ x ] \ rlx & b := [ y ] \rlx \\
\hline
[ y ] \rlx := 1 & c := [ x ] \rlx & \hline
\end{array} \]

\[ \{ \ldots, (a=b=c=1), \ldots \} \]
Weak behavior can be controlled with access modes

<table>
<thead>
<tr>
<th></th>
<th>[x] = [y] = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x]_{sc} := 1</td>
<td>a := [x]_{sc}</td>
</tr>
<tr>
<td>[y]_{rlx} := 1</td>
<td>b := [y]_{rlx}</td>
</tr>
<tr>
<td>[y]_{rlx} := 1</td>
<td>c := [x]_{sc}</td>
</tr>
</tbody>
</table>
Weak behavior can be controlled with access modes

\[ x = y = 0 \]

\[ x_{sc} := 1 \]
\[ a := x_{sc} \]
\[ b := y_{rlx} \]

\[ y_{rlx} := 1 \]
\[ c := x_{sc} \]
Weak behavior can be controlled with access modes but the effect is not obvious

\[
\begin{array}{|c|c|c|}
\hline
[x] = [y] = 0 & [x]^{sc} := 1 & a := [x]^{sc} \\
\hline
[y]^{rlx} := 1 & b := [y]^{rlx} & c := [x]^{sc} \\
\hline
\end{array}
\]

\[a = b = 1, c = 0\]
C++ solution: strengthen access mode everywhere

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$[x] = [y] = 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[x]^{sc} := 1$</td>
<td>$a := [x]^{sc}$</td>
<td>$b := [y]^{sc}$</td>
</tr>
<tr>
<td>$[y]^{sc} := 1$</td>
<td></td>
<td>$c := [x]^{sc}$</td>
</tr>
<tr>
<td></td>
<td>$a = b = 1, c = 0$</td>
<td></td>
</tr>
</tbody>
</table>
OCaml MM: reasonable rules for racy programs

\[
\begin{array}{|c|c|c|}
\hline
& [x] = [y] = 0 & \\
\hline
[x]^{at} := 1 & a := [x]^{at} & b := [y]^{na} \\
\hline
[y]^{na} := 1 & c := [x]^{at} & \\
\hline
\end{array}
\]
OCaml MM: reasonable rules for racy programs

\[
\begin{array}{c|c|c}
[\mathbf{x}] = [\mathbf{y}] = 0 \\
[\mathbf{x}]^{at} := 1 & a := [\mathbf{x}]^{at} & b := [\mathbf{y}]^{na} \\
[\mathbf{y}]^{na} := 1 & c := [\mathbf{x}]^{at}
\end{array}
\]
**OCaml MM: reasonable rules for racy programs**

<table>
<thead>
<tr>
<th></th>
<th>$[x] = [y] = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[x]^{at} := 1$</td>
<td>$a := [x]^{at}$</td>
</tr>
<tr>
<td>$[y]^{na} := 1$</td>
<td>$b := [y]^{na}$</td>
</tr>
<tr>
<td>$[y]^{na} := 1$</td>
<td>$c := [x]^{at}$</td>
</tr>
</tbody>
</table>
OCaml MM: reasonable rules for racy programs

\[
\begin{align*}
\text{[x]} &= \text{[y]} = 0 \\
\text{[x]}_{\text{at}} := 1 & \quad \text{a} := \text{[x]}_{\text{at}} \\
\text{[y]}_{\text{na}} := 1 & \quad \text{c} := \text{[x]}_{\text{at}} \\
\end{align*}
\]
OCaml MM: reasonable rules for racy programs

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

\[ [x]^{at} := 1 \quad a := [x]^{at} \]

\[ [y]^{na} := 1 \quad b := [y]^{na} \]

\[ [y]^{na} := 1 \quad c := [x]^{at} \]

Local data race freedom: result of reading x doesn’t depend on the race on y
OCaml MM: reasonable rules for racy programs

| [x] = [y] = 0 | [x]at := 1 | a := [x]at | b := [y]na | [y]na := 1 | c := [x]at |

Local data race freedom: result of reading x doesn’t depend on the race on y.
OCaml MM: reasonable rules for racy programs

Local data race freedom: result of reading $x$ doesn't depend on the race on $y$
OCaml MM: reasonable rules for racy programs

<table>
<thead>
<tr>
<th>[x] = [y] = 0</th>
<th>[x]at := 1</th>
<th>a := [x]at</th>
<th>b := [y]na</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[y]na := 1</td>
<td></td>
<td>c := [x]at</td>
</tr>
</tbody>
</table>

Local data race freedom: result of reading x doesn’t depend on the race on y
OCaml MM: reasonable rules for racy programs

Local data race freedom:
result of reading x doesn't depend on the race on y

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

\[ [x]^{at} := 1 \quad a := [x]^{at} \quad b := [y]^{na} \]

\[ [y]^{na} := 1 \quad c := [x]^{at} \]

\[ a = b = 1, c = 0 \]
OCaml MM guarantees should be implemented

\[
\begin{array}{|c|c|c|}
\hline
[x] = [y] = 0 \\
[x]^{sc} := 1 & a := [x]^{sc} & b := [y]^{rlx} \\
[y]^{rlx} := 1 & c := [x]^{sc} \\
\hline
\end{array}
\]

\[a = b = 1, \ c = 0\]
OCaml MM guarantees should be implemented

\[ \text{[compile(Prog)]}_{\text{CPU}} \]
OCaml MM guarantees should be implemented by providing a correct compilation scheme
We’ve proved compilation correctness from OCaml MM to Power

OCaml MM

x86

ARM
We’ve proved compilation correctness from OCaml MM to Power
We’ve proved compilation correctness from OCaml MM to Power using IMM
Another execution representation is needed

<table>
<thead>
<tr>
<th>[x] = [y] = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x] := 1</td>
</tr>
<tr>
<td>[y] := 1</td>
</tr>
</tbody>
</table>

a = b = 1, c = 0
Consider the execution as a graph

\[
\begin{array}{c|c|c}
\text{[x]} &=& \text{[y]} = 0 \\
\text{[x]}^\text{at} := 1 & a := [x]^\text{at} & b := [y]^\text{na} \\
\text{[y]}^\text{na} := 1 & c := [x]^\text{at} \\
a = b = 1, c = 0
\end{array}
\]
A permission of execution is determined by its graph

\[ x = y = 0 \]

\[ x^\text{at} := 1 \quad a := [x]^\text{at} \quad b := [y]^\text{na} \]

\[ [y]^\text{na} := 1 \quad c := [x]^\text{at} \]
A permission of execution is determined by its graph

<table>
<thead>
<tr>
<th>[x] = [y] = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x]at := 1</td>
</tr>
<tr>
<td>[y]na := 1</td>
</tr>
</tbody>
</table>

OCaml MM: \( a = b = 1, c = 0 \)

OMM: no cycles made of \( \text{po}, \text{rf} \) and \( \text{rb} \)
Compilation correctness in terms of graphs

\[ \text{compile}(\text{Prog}) \]_{\text{IMM}}

\[ \text{Prog} \]_{\text{OCaml MM}}
Compilation correctness in terms of graphs

\[ \text{compile}(\text{Prog}) \]_{\text{IMM}}

\[ \text{Graphs}_{\text{IMM}}(\text{compile}(\text{Prog})) \]

\[ \text{Prog} \]_{\text{OCaml MM}}

\[ \text{Graphs}_{\text{OCaml MM}}(\text{Prog}) \]
The identity compilation scheme won’t work

\[
\text{compile}(\text{Prog}) = [\text{na} \rightarrow \text{rlx}, \text{at} \rightarrow \text{sc}]\text{Prog}
\]
The identity compilation scheme won’t work

\[
\text{compile}(\text{Prog}) = [\text{na} \rightarrow \text{rlx}, \text{at} \rightarrow \text{sc}]\text{Prog}
\]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>([x] = [y] = 0)</td>
<td>([x]^{\text{sc}} := 1)</td>
<td>(a := [x]^{\text{sc}})</td>
</tr>
<tr>
<td>([y]^{\text{rlx}} := 1)</td>
<td>(b := [y]^{\text{rlx}})</td>
<td>(c := [x]^{\text{sc}})</td>
</tr>
</tbody>
</table>
The identity compilation scheme won’t work

\[
\text{compile(Prog) = \{na \rightarrow \text{rlx}, at \rightarrow \text{sc}\}Prog}
\]

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

<table>
<thead>
<tr>
<th>[x]^{sc} := 1</th>
<th>a := [x]^{sc}</th>
<th>b := [y]^{rlx}</th>
</tr>
</thead>
<tbody>
<tr>
<td>[y]^{rlx} := 1</td>
<td>c := [x]^{sc}</td>
<td></td>
</tr>
</tbody>
</table>

\[
W^{sc}(x, 1) \quad W^{rlx}(y, 1) \quad R^{sc}(x, 0)
\]

\[
R^{sc}(x, 1) \quad R^{rlx}(y, 1)
\]
The identity compilation scheme won’t work

\[
\text{compile}(\text{Prog}) = [\text{na} \rightarrow \text{rlx}, \text{at} \rightarrow \text{sc}]\text{Prog}
\]

\[
\begin{array}{|c|c|c|}
\hline
[x] &=& [y] = 0 \\
[x]^\text{sc} &:=& 1 \\
[y]^\text{rlx} &:=& 1 \\
a &=& [x]^\text{sc} \\
b &=& [y]^\text{rlx} \\
c &=& [x]^\text{sc} \\
\hline
\end{array}
\]

\[
a = b = 1, c = 0
\]

IMM: can have a cycle made of \textbf{po}, \textbf{rf} and \textbf{rb}
The identity compilation scheme won’t work

$$\text{Graphs}_{\text{OCaml MM}}(\text{Prog})$$

$$\text{Graphs}_{\text{IMM}}(\text{compile(Prog)})$$
Observed writes should remain so

\[ w^{sc}(x, 1), \quad w^{rlx}(y, 1), \quad R^{sc}(x, 0), \quad R^{rlx}(y, 1) \]
Observed writes should remain so

\[ x = 1 \checkmark \]

\[ W^{sc}(x, 1) \quad R^{sc}(x, 1) \quad R^{rlx}(y, 1) \quad R^{sc}(x, 0) \]

\[ W^{rlx}(y, 1) \quad r_{f} \quad r_{f} \]

\[ rb \]
Observed writes should remain so

\[ \mathcal{W}_{sc}(x, 1) \rightarrow R^{sc}(x, 1) \rightarrow R^{rlx}(y, 1) \]

\[ \mathcal{W}_{rlx}(y, 1) \rightarrow R^{rlx}(y, 1) \rightarrow R^{sc}(x, 0) \]

\[ x = 1 \checkmark \]

\[ x = 0 \times \]
Observed writes should remain so
= graph should have no cycles with rb

\[ x = 1 \checkmark \]

\[ R^{sc}(x, 1) \]
\[ R^{rlx}(y, 1) \]

\[ W^{sc}(x, 1) \]
\[ W^{rlx}(y, 1) \]
\[ R^{sc}(x, 0) \]

\[ x = 0 \times \]
“Release” known writes and “acquire” them next.
“Release” known writes and “acquire” them next

\[ R^{sc}(x, 1) \]  \[ R^{rlx}(y, 1) \]

\[ W^{sc}(x, 1) \]  \[ W^{rlx}(y, 1) \]  \[ R^{sc}(x, 1) \]
“Release” known writes and “acquire” them next

\[ x = 1 \]

\[ R^{sc}(x, 1) \quad R^{rlx}(y, 1) \]

\[ W^{sc}(x, 1) \quad W^{rlx}(y, 1) \quad R^{sc}(x, 1) \]

rf

rf

rf
“Release” known writes and “acquire” them next

\[ x = 1 \]

[ ✓ ]

[ RF ]

\[ R^{sc}(x, 1) \quad R^{rlx}(y, 1) \quad x = 1 \quad [ ✓ ] \]

\[ W^{sc}(x, 1) \quad W^{rlx}(y, 1) \quad R^{sc}(x, 1) \quad [ RF ] \]
Implemented with release and acquire fences
Implemented with release and acquire fences

\[ \text{compile}(\text{Prog}) = [\text{na} \to \text{rlx}, \text{at} \to \text{sc}]\text{Prog} + \text{Fences}^{\text{rel}} + \text{Fences}^{\text{acq}} \]
Implemented with release and acquire fences

\[
\text{compile}(\text{Prog}) = [\text{na} \rightarrow \text{rlx}, \text{at} \rightarrow \text{sc}]\text{Prog} + \text{Fences}^{\text{rel}} + \text{Fences}^{\text{acq}}
\]

<table>
<thead>
<tr>
<th>([x] = [y] = 0)</th>
<th>([x]^{\text{sc}} := 1)</th>
<th>(a := [x]^{\text{sc}})</th>
<th>(b := [y]^{\text{rlx}})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\text{fence}^{\text{rel}})</td>
<td>(\text{fence}^{\text{acq}})</td>
<td></td>
</tr>
<tr>
<td>([y]^{\text{rlx}} := 1)</td>
<td></td>
<td></td>
<td>(c := [x]^{\text{sc}})</td>
</tr>
</tbody>
</table>
Implemented with release and acquire fences

\[
\text{compile}(\text{Prog}) = [\text{na} \to \text{rlx}, \text{at} \to \text{sc}]\text{Prog} + \text{Fences}^{\text{rel}} + \text{Fences}^{\text{acq}}
\]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[x] = 0</td>
<td>[y] = 0</td>
<td></td>
</tr>
<tr>
<td>[x]_{\text{sc}} := 1</td>
<td>a := [x]_{\text{sc}}</td>
<td>b := [y]_{\text{rlx}}</td>
</tr>
<tr>
<td>fence^{\text{rel}}</td>
<td>fence^{\text{acq}}</td>
<td></td>
</tr>
<tr>
<td>[y]_{\text{rlx}} := 1</td>
<td>c := [x]_{\text{sc}}</td>
<td></td>
</tr>
</tbody>
</table>
Implemented with release and acquire fences

\[
\text{compile}(\text{Prog}) = [\text{na} \rightarrow \text{rlx}, \text{at} \rightarrow \text{sc}]\text{Prog} + \text{Fences}^{\text{rel}} + \text{Fences}^{\text{acq}}
\]

\[
\begin{array}{|c|c|c|}
\hline
[x] = [y] = 0 \\
\hline
[x]^\text{sc} := 1 & a := [x]^\text{sc} & b := [y]^\text{rlx} \\
\hline
\text{fence}^{\text{rel}} & \text{fence}^{\text{acq}} \\
\hline
[y]^\text{rlx} := 1 & c := [x]^\text{sc} \\
\hline
\hline
\end{array}
\]

\[a = b = 1, c = 0\]

IMM: can have a cycle made of \textbf{po}, \textbf{rf} and \textbf{rb} if there is \textbf{rf} without \textbf{sc} and fences
An IMM-inconsistent behavior is now prohibited

Graphs \(\text{OCaml MM}(\text{Prog})\)

Graphs \(\text{IMM}(\text{compile(Prog)})\)
The resulting scheme prohibits unwanted behaviors

<table>
<thead>
<tr>
<th>OCaml MM</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r := [x]\text{na} )</td>
<td>( r := [x]\text{rlx} )</td>
</tr>
<tr>
<td>( [x]\text{na} := v )</td>
<td>( \text{fence}^{\text{acqrel}}; [x]\text{rlx} := v )</td>
</tr>
<tr>
<td>( r := [x]\text{at} )</td>
<td>( \text{fence}^{\text{acq}}; r := [x]\text{sc} )</td>
</tr>
<tr>
<td>( [x]\text{at} := v )</td>
<td>( \text{fence}^{\text{acq}}; \text{exchange}^{\text{sc}}(x, v) )</td>
</tr>
</tbody>
</table>
Takeaway

● Compilation scheme from OCaml MM to IMM
● Proved to be correct
● Formalized in Coq

Machine-verified compilation scheme from OCaml MM to Power

https://github.com/weakmemory/imm