Weak Memory

Anton Podkopaev

St. Petersburg University, JetBrains Research, Russia

13.12.2017
Memory models. What, why, and when to care

CPU and PL MMs. Promise MM

Compilation correctness for Promise MM

Agenda
Agenda

- Memory models. What, why, and when to care
Agenda

- Memory models. What, why, and when to care

- CPU and PL MMs. Promise MM
Agenda

- Memory models. What, why, and when to care
- CPU and PL MMs. Promise MM
- Compilation correctness for Promise MM to \{x86, Power, ARM\}
Agenda

- Memory models. What, why, and when to care

- CPU and PL MMs. Promise MM

- Compilation correctness for Promise MM to \{x86, Power, ARM\}
Memory models. What, why, and when to care

CPU and PL MMs. Promise MM

Compilation correctness for Promise MM to \{x86, Power, ARM\}
Memory model (MM) is concurrent system’s semantics
Sequential consistency
[Lamport, 1979]
Sequential consistency 
[Lamport, 1979]

system’s behaviors —
program interleavings
Execution in SC

\[ [x] := 1; \]
\[ a := [y]; \]
\[ b := [x]; \]

Values
\[ a = \perp; b = \perp \]

Memory
\[ [x] \leftarrow 0; [y] \leftarrow 0 \]
Execution in SC

\[
\begin{align*}
[x] &:= 1; \\
 a &:= [y]; & \quad [y] &:= 1; \\
 b &:= [x];
\end{align*}
\]

**Values**

\[
a = \bot; \quad b = \bot
\]

**Memory**

\[
[x] \leftarrow 1; \quad [y] \leftarrow 0
\]
Execution in SC

\[
\begin{align*}
[x] & := 1; & [y] & := 1; \\
\frac{[x]}{a} & := [y]; & \frac{[y]}{b} & := [x];
\end{align*}
\]

Values
\[
\begin{align*}
a &= \perp; \\
b &= \perp
\end{align*}
\]

Memory
\[
\begin{align*}
[x] & \leftarrow 1; \\
[y] & \leftarrow 1
\end{align*}
\]
Execution in SC

\[
\begin{align*}
[x] & := 1; \\
\quad \quad a := [y]; \\
\quad \quad b := [x]; \\
\end{align*}
\]

Values
\[
a = \perp; \ b = 1
\]

Memory
\[
[x] \leftarrow 1; \ [y] \leftarrow 1
\]
Execution in SC

\[
\begin{align*}
[x] & := 1; \\
  a & := [y]; \\
[y] & := 1; \\
  b & := [x]; \\
\end{align*}
\]

Values:
\[a = 1; b = 1\]

Memory:
\[x \leftarrow 1; y \leftarrow 1\]
Execution in SC

\[
\begin{align*}
[x] & := 1; & [y] & := 1; \\
a & := [y]; & b & := [x];
\end{align*}
\]

Impossible to get \( a = b = 0 \)

Values
\[
a = 1; \quad b = 1
\]

Memory
\[
[x] \leftarrow 1; \quad [y] \leftarrow 1
\]
Is it the same in reality?
Is it the same in reality?
Let’s check!
Execution in SC

\[
\begin{align*}
[x] & := 1; \\
    a & := [y]; \\
\end{align*}
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \cdot
Modern Compilers and CPUs are Optimizing
Modern Compilers and CPUs are Optimizing

Features

- reorderings
- cache
- buffers
- read-after-write elimination
- speculative execution
- fake dependency elimination
- ...
Modern Compilers and CPUs are Optimizing

Features

- reorderings
- cache
- buffers
- read-after-write elimination
- speculative execution
- fake dependency elimination
- ...
Modern Compilers and CPUs are Optimizing

Features

- reorderings
- cache
- buffers
- read-after-write elimination
- speculative execution
- fake dependency elimination
- ...

Correct for one thread
Modern Compilers and CPUs are Optimizing

Features

- reorderings
- cache
- buffers
- read-after-write elimination
- speculative execution
- fake dependency elimination
- ...

Lead to strange concurrent behaviors

Correct for one thread
Modern Compilers and CPUs are Optimizing

Features

- reorderings
- cache
- buffers
- read-after-write elimination
- speculative execution
- fake dependency elimination
- ... 

Correct for **one** thread

Lead to **weak** concurrent behaviors
Non-SC behaviors are called weak
Non-SC behaviors are called **weak**

**Weak** MMs allow weak behaviors
Non-SC behaviors are called **weak**

**Weak** MMs allow weak behaviors

Real systems have weak MMs
Non-SC behaviors are called **weak**

**Weak** MMs allow weak behaviors

Real systems have weak MMs
(i.e., x86, Power, ARM, C++, Java)
Realistic weak MMs are subtle
...and different to each other
But most have *data race freedom* (DRF) results
But most have \textit{data race freedom} (DRF) results:

No data races $\Rightarrow$ only SC behaviors
When **not** to care about Weak MMs

Writing/verifying a program, which

- has immutable data only
- is single-threaded
- is properly locked multi-threaded
When to care about Weak MMs

Writing/verifying lock-free code (i.e., locks themselves)
Agenda

- Memory models. What, why, and when to care
- CPU and PL MMs. Promise MM
- Compilation correctness for Promise MM to \{x86, Power, ARM\}
[Store Buffering in x86-TSO
[Owens et al., 2009]]
Load Buffering

\[
\begin{align*}
a & := [x]; & \quad b & := [y]; \\
y & := 1; & \quad x & := 1;
\end{align*}
\]

Final values \(a = 1, b = 1\)
Load Buffering in ARMv8 POP

\[
\begin{align*}
a & := [x]; \\
[y] & := 1; \\

\end{align*}
\]

\[
\begin{align*}
b & := [y]; \\
x & := 1; \\
[y] & := 1; \\
x & := 1; \\
\end{align*}
\]

Final values \( a = \_ \), \( b = \_ \)
Load Buffering in ARMv8 POP

\[
\begin{align*}
  a & := [x]; \\
  [y] & := 1; \\
  b & := [y]; \\
  [x] & := 1;
\end{align*}
\]

Memory

\[
[x] \leftarrow 0; [y] \leftarrow 0
\]

Final values \( a = \_ \), \( b = \_ \)
Load Buffering in ARMv8 POP

a := [x];
[y] := 1;

b := [y];
x := 1;

b ← [y]

[x] ← 0; [y] ← 0

Final values a = __, b = __
Load Buffering in ARMv8 POP

\[
\begin{align*}
  a &= [x]; \\
  [y] &= 1; \\
  b &= [y]; \\
  [x] &= 1;
\end{align*}
\]

Memory
\[
[x] \leftarrow 0; [y] \leftarrow 0
\]

Final values \( a = \_ \), \( b = \_ \)
Load Buffering in ARMv8 POP

 Independently:

\[
\begin{align*}
a & := [x]; \\
[y] & := 1;
\end{align*}
\]

\[
\begin{align*}
b & := [y]; \\
[x] & := 1;
\end{align*}
\]

\[\begin{align*}
[x] & \leftarrow 1 \\
b & \leftarrow [y]
\end{align*}\]

Memory:

\[
[x] \leftarrow 0; [y] \leftarrow 0
\]

Final values: \(a = \_\), \(b = \_\)
Load Buffering in ARMv8 POP

\[ a := [x]; \]
\[ [y] := 1; \]

\[ b := [y]; \]
\[ [x] := 1; \]

Final values: \( a = \_ , b = \_ \)
Load Buffering in ARMv8 POP

\[
\begin{align*}
  a & := [x]; \\
  [y] & := 1;
\end{align*}
\]

\[
\begin{align*}
  b & := [y]; \\
  [x] & := 1;
\end{align*}
\]

\[
b \leftarrow [y]
\]

Memory

\[
[x] \leftarrow 1; [y] \leftarrow 0
\]

Final values \( a = \_ \), \( b = \_ \)
Load Buffering in ARMv8 POP

\[
\begin{align*}
    a & := [x]; \\
    b & := [y]; \\
    [y] & := 1; \\
    [x] & := 1; \\
    [y] & \leftarrow 1 \\
    b & \leftarrow [y]
\end{align*}
\]

Memory
\[
[x] \leftarrow 1; [y] \leftarrow 0
\]

Final values \( a = \_ \), \( b = \_ \)
Load Buffering in ARMv8 POP

\[ a := [x]; b := [y]; \]
\[ [y] := 1; [x] := 1; \]

\[ b \leftarrow [y] \]

\[ [x] \leftarrow 1; [y] \leftarrow 1 \]

Final values \( a = \_ \), \( b = \_ \)
Load Buffering in ARMv8 POP

\[
\begin{align*}
    a & := [x]; \\
    [y] & := 1;
\end{align*}
\]

\[
\begin{align*}
    b & := [y]; \\
    [x] & := 1;
\end{align*}
\]

Memory
\[
[x] \leftarrow 1; [y] \leftarrow 1
\]

Final values \(a = \_, b = 1\)
Load Buffering in ARMv8 POP

\[ a := [x]; \]
\[ [y] := 1; \]
\[ b := [y]; \]
\[ [x] := 1; \]

\[ a \leftarrow [x] \]

Memory
\[ [x] \leftarrow 1; [y] \leftarrow 1 \]

Final values \( a = \_, b = 1 \)
Load Buffering in ARMv8 POP

\[
\begin{align*}
a & := [x]; \\
y & := 1;
\end{align*}
\]

\[
\begin{align*}
b & := [y]; \\
x & := 1;
\end{align*}
\]

\[
[x] \leftarrow 1; [y] \leftarrow 1
\]

Final values \(a = 1, \ b = 1\)
ARM-Weak in ARMv8 POP

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

\[a = 1?\]
ARM-Weak in ARMv8 POP

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

Values

\[
\begin{align*}
  a &= \bot \\
  b &= \bot \\
  c &= \bot
\end{align*}
\]

Memory

\[
[x] \leftarrow 0; [y] \leftarrow 0
\]
ARM-Weak in ARMv8 POP

Values

\[
\begin{align*}
a &= \bot \\
b &= \bot \\
c &= \bot
\end{align*}
\]

Memory

\[
\begin{align*}
[x] &\leftarrow 0; [y] \leftarrow 0
\end{align*}
\]
ARM-Weak in ARMv8 POP

Values
- $a = \perp$
- $b = \perp$
- $c = \perp$

Memory
- $[x] \leftarrow 0; [y] \leftarrow 0$

Expressions:
- $a := [x]$
- $[x] := 1$
- $b := [x]$
- $[y] := b$
- $c := [y]$
- $[x] := c$
ARM-Weak in ARMv8 POP

Values

\[ a = \bot \]
\[ b = \bot \]
\[ c = \bot \]

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

\[
\begin{align*}
a & \leftarrow [x] \\
[x] & \leftarrow 0; [y] \leftarrow 0
\end{align*}
\]
ARM-Weak in ARMv8 POP

Values

\[ a = \bot \]
\[ b = \bot \]
\[ c = \bot \]

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

\[
\begin{align*}
  [x] & \leftarrow 1 \\
  a & \leftarrow [x] \\
  [x] & \leftarrow 0; [y] \leftarrow 0
\end{align*}
\]
ARM-Weak in ARMv8 POP

Values
\[
\begin{align*}
  a &= \bot \\
  b &= \bot \\
  c &= \bot
\end{align*}
\]

Memory
\[
\begin{align*}
  [x] &\leftarrow 0; \ [y] &\leftarrow 0
\end{align*}
\]
ARM-Weak in ARMv8 POP

Values

\[ a = \bot \]
\[ b = \bot \]
\[ c = \bot \]

Memory

\[ [x] \leftarrow 0; [y] \leftarrow 0 \]
ARM-Weak in ARmv8 POP

Values

\[a = \bot,\]
\[b = \bot,\]
\[c = \bot.\]

Memory

\[\text{[x]} \leftarrow 0; \text{[y]} \leftarrow 0.\]
ARM-Weak in ARMv8 POP

Values

\[ a = \bot \]
\[ b = \bot \]
\[ c = \bot \]

Memory

\[ [x] \leftarrow 0; [y] \leftarrow 0 \]
ARM-Weak in ARMv8 POP

**Values**
- \(a = \perp\)
- \(b = 1\)
- \(c = \perp\)

**Memory**
- \([x] \leftarrow 0; [y] \leftarrow 0\)

**Statements**
- \(a := [x];\)
- \([x] := 1;\)
- \(b := [x];\)
- \([y] := b;\)
- \(c := [y];\)
- \([x] := c;\)
ARM-Weak in ARMv8 POP

Values
\[a = \perp\]
\[b = 1\]
\[c = \perp\]

Memory
\[[x] \leftarrow 0; [y] \leftarrow 0\]

\[\begin{align*}
a & := [x]; \\
[x] & := 1;
\end{align*}\]

\[\begin{align*}
b & := [x]; \\
[y] & := b;
\end{align*}\]

\[\begin{align*}
c & := [y]; \\
[x] & := c;
\end{align*}\]
**ARM-Weak in ARMv8 POP**

Values:
- \( a = \bot \)
- \( b = 1 \)
- \( c = \bot \)

Memory:
\[ [x] \leftarrow 0; [y] \leftarrow 0 \]
ARM-Weak in ARMv8 POP

Values

\[ a = \perp \]
\[ b = 1 \]
\[ c = \perp \]

Independent

\[ a \leftarrow [x] \]
\[ b \leftarrow [x] \]
\[ c \leftarrow [y] \]

Memory

\[ [x] \leftarrow 0; [y] \leftarrow 0 \]
ARM-Weak in ARMv8 POP

Values

\(a = \bot\)
\(b = 1\)
\(c = \bot\)

Memory

\([x] \leftarrow 0; [y] \leftarrow 0\)
ARM-Weak in ARMv8 POP

Values

\[
\begin{align*}
  a &= \bot \\
  b &= 1 \\
  c &= \bot
\end{align*}
\]

\[
\begin{align*}
  [x] &:= 1; \\
  [y] &:= b; \\
  [x] &:= c;
\end{align*}
\]

\[
\begin{align*}
  [x] &\leftarrow 1 \\
  [y] &\leftarrow 1 \\
  a &\leftarrow [x]
\end{align*}
\]

Independent

\[
\begin{align*}
  [x] &\leftarrow 0; [y] &\leftarrow 0
\end{align*}
\]
ARM-Weak in ARMv8 POP

Values

\[ a = \bot \]
\[ b = 1 \]
\[ c = \bot \]

Memory

\[ [x] \leftarrow 0; [y] \leftarrow 0 \]
ARM-Weak in ARMv8 POP

Values

\[ a = \bot \]
\[ b = 1 \]
\[ c = \bot \]

\[ a := [x]; \]
\[ x := 1; \]
\[ b := [x]; \]
\[ y := b; \]
\[ c := [y]; \]
\[ x := c; \]

\[ x \leftarrow 1 \]
\[ a \leftarrow [x] \]

\[ x \leftarrow 0; y \leftarrow 1 \]
ARM-Weak in ARMv8 POP

Values

\[ a = \bot \]
\[ b = 1 \]
\[ c = \bot \]

\[
\begin{align*}
a &:= [x]; \\
[x] &:= 1;
\end{align*}
\]

\[
\begin{align*}
b &:= [x]; \\
[y] &:= b;
\end{align*}
\]

\[
\begin{align*}
c &:= [y]; \\
[x] &:= c;
\end{align*}
\]

\[ c \leftarrow [y] \]

Memory

\[
\begin{align*}
[x] &\leftarrow 0; \\
y &\leftarrow 1
\end{align*}
\]
ARM-Weak in ARMv8 POP

Values

\[
\begin{align*}
a &= \bot \\
b &= 1 \\
c &= 1
\end{align*}
\]

\[
\begin{align*}
a &:= [x]; \\
[x] &:= 1;
\end{align*}
\]

\[
\begin{align*}
b &:= [x]; \\
[y] &:= b;
\end{align*}
\]

\[
\begin{align*}
c &:= [y]; \\
[x] &:= c;
\end{align*}
\]

\[
\begin{align*}
[x] &\leftarrow 1 \\
a &\leftarrow [x]
\end{align*}
\]

\[
\begin{align*}
[x] &\leftarrow 0; \\
[y] &\leftarrow 1
\end{align*}
\]
ARM-Weak in ARMv8 POP

Values

\[a = \perp\]
\[b = 1\]
\[c = 1\]

Memory

\[[x] \leftarrow 0; [y] \leftarrow 1\]
ARM-Weak in ARMv8 POP

Values

\[ a = \perp \]
\[ b = 1 \]
\[ c = 1 \]

Memory

\[ [x] \leftarrow 1; [y] \leftarrow 1 \]
ARM-Weak in ARMv8 POP

Memory models. What, why, and when to care

Values

\[
\begin{align*}
a &= 1 \\
b &= 1 \\
c &= 1
\end{align*}
\]

[\[x\] := 1; \[y\] := \[x\]; \[x\] := \[c\];]

\[
\begin{align*}
a &= [x]; \\
b &= [x]; \\
c &= [y]; \\
[x] &\leftarrow 1; \[y\] &\leftarrow 1
\end{align*}
\]
CPU MM should:
CPU MM should:

1. describe real CPUs
CPU MM should:

1. describe real CPUs
2. save room for future optimizations
CPU MM should:

1. describe real CPUs
2. save room for future optimizations
3. provide reasonable guarantees for PLs
MM for PL?
MM for PL?
Has to satisfy 3 requirements
1. Efficient Compilation
1. Efficient Compilation

\[
\begin{align*}
[x] & := 1; & [y] & := 1; \\
 a & := [y]; & b & := [x];
\end{align*}
\]
1. Efficient Compilation

Source (SC MM)

\[
\begin{align*}
[x] & := 1; & [y] & := 1; \\
a & := [y]; & b & := [x];
\end{align*}
\]
1. Efficient Compilation

Source (SC MM)

\[
[x] := 1; \quad [y] := 1; \\
a := [y]; \quad b := [x];
\]

Target (x86 MM)
1. Efficient Compilation

Source (SC MM)

\[
[x] := 1; \\
\quad a := [y]; \\
\quad [y] := 1; \\
\quad b := [x];
\]

Target (x86 MM)

\[
[x] := 1; \\
\quad mfence; \\
\quad [y] := 1; \\
\quad mfence; \\
\quad a := [y]; \\
\quad b := [x];
\]
1. Efficient Compilation

Source (SC MM)

\[
\begin{align*}
[x] & := 1; \\
a & := [y]; \\
\end{align*}
\]

Target (x86 MM)

\[
\begin{align*}
[x] & := 1; \\
mfence; \\
a & := [y]; \\
\end{align*}
\]

\[
\begin{align*}
[y] & := 1; \\
\end{align*}
\]

\[
\begin{align*}
b & := [x]; \\
\end{align*}
\]

Not efficient
2. Compiler Optimizations
2. Compiler Optimizations

\[ x := 1; \quad \| \quad y := 1; \]

\[ a := [y]; \quad \| \quad b := [x]; \]
2. Compiler Optimizations

Original

\[
\begin{align*}
[x] & := 1; & [y] & := 1; \\
\text{a} & := \text{[y]}; & \text{b} & := \text{[x]};
\end{align*}
\]
2. Compiler Optimizations

Original

\[
\begin{align*}
[x] & := 1; \\
[y] & := 1; \\
a & := [y] ; \\
b & := [x];
\end{align*}
\]

Optimized

\[
\begin{align*}
[a] & := 1; \\
[y] & := 1; \\
[x] & := 1; \\
b & := [x];
\end{align*}
\]
2. Compiler Optimizations

Original

\[
\begin{align*}
x &:= 1; \quad [y] := 1; \\
a &= [y]; \\
b &= [x];
\end{align*}
\]

Optimized

\[
\begin{align*}
a &= [y]; \\
[y] &= 1; \\
[x] &= 1; \\
b &= [x];
\end{align*}
\]
3. No Out-Of-Thin-Air
3. No Out-Of-Thin-Air

\[
\begin{align*}
a &:= [x]; & b &:= [y]; \\
[y] &:= a; & [x] &:= b;
\end{align*}
\]
3. No Out-Of-Thin-Air

\[
\begin{align*}
a & := [x]; & b & := [y]; \\
[y] & := a; & [x] & := b;
\end{align*}
\]

C/C++11 MM allows \( a = b = 42 \)
Memory Models for PLs
Memory Models for PLs

SC MM, [Lamport, 1979]
Java MM, [Manson et al., 2005]
C/C++11 MM, [Batty et al., 2011]
Memory Models for PLs

SC MM, [Lamport, 1979]  EC
Java MM, [Manson et al., 2005]  ✓
C/C++11 MM, [Batty et al., 2011]  ✓

Requirements:

• allow Efficient Compilation (x86, Power, ARM)
• validate Compiler Optimizations (merging, rearranging, etc)
• no Out-Of-Thin-Air
Memory Models for PLs

SC MM, [Lamport, 1979]  
Java MM, [Manson et al., 2005]  
C/C++11 MM, [Batty et al., 2011]

<table>
<thead>
<tr>
<th></th>
<th>EC</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Requirements:

- allow Efficient Compilation (x86, Power, ARM)
- validate Compiler Optimizations (merging, rearranging, etc)
- no Out-Of-Thin-Air
## Memory Models for PLs

<table>
<thead>
<tr>
<th>Memory Models</th>
<th>EC</th>
<th>CO</th>
<th>No OOTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC MM, [Lamport, 1979]</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Java MM, [Manson et al., 2005]</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>C/C++11 MM, [Batty et al., 2011]</td>
<td>✓</td>
<td>✗*</td>
<td>✗</td>
</tr>
</tbody>
</table>

### Requirements:
- allow **Efficient Compilation** (x86, Power, ARM)
- validate **Compiler Optimizations** (merging, rearranging, etc)
- no **Out-Of-Thin-Air**
Memory Models for PLs

SC MM, [Lamport, 1979]
Java MM, [Manson et al., 2005]
C/C++11 MM, [Batty et al., 2011]

Requirements:
- allow Efficient Compilation (x86, Power, ARM)
- validate Compiler Optimizations (merging, rearranging, etc)
- no Out-Of-Thin-Air

<table>
<thead>
<tr>
<th>EC</th>
<th>CO</th>
<th>No OOTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓*</td>
<td>✗</td>
</tr>
</tbody>
</table>
C/C++11 MM has OOTA
C/C++11 MM has OOTA. Why?
C/C++11 MM is **axiomatic** MM
Load Buffering in C/C++11

\[
\begin{align*}
  a & := [x]; & b & := [y]; \\
  [y] & := 1; & [x] & := 1;
\end{align*}
\]
Load Buffering in C/C++11

\[
\begin{align*}
    a & := [x]; & b & := [y]; \\
    [y] & := 1; & [x] & := 1;
\end{align*}
\]
Load Buffering in C/C++11

\[
\begin{align*}
    a &:= [x]; & b &:= [y]; \\
    [y] &:= 1; & [x] &:= 1;
\end{align*}
\]
Load Buffering in C/C++11

\[ a := [x]; \quad b := [y]; \quad [y] := 1; \quad [x] := 1; \]

Axioms:

1. \textit{hb} is acyclic

...
Load Buffering in C/C++11

\[
a := [x]; \quad b := [y];
\]
\[
[y] := a; \quad [x] := b;
\]

Axioms:
1. \(hb\) is acyclic

\[
\begin{align*}
R_{x1} & \quad R_{y1} \\
po \downarrow & \quad po \\
W_{y1} & \quad W_{x1}
\end{align*}
\]

\[
\begin{array}{c}
\text{rf} \\
r \downarrow
\end{array}
\]
Load Buffering in C/C++11

\[ a := [x]; \quad b := [y]; \]
\[ [y] := a; \quad [x] := b; \]

Axioms:

1. \textbf{hb} is acyclic

\[ \begin{array}{c}
R_{x8} \quad R_{y8} \\
po \downarrow \quad po \downarrow \\
W_{y8} \quad W_{x8}
\end{array} \]

\[ \begin{array}{c}
R_{x8} \quad R_{y8} \\
po \downarrow \quad po \downarrow \\
W_{y8} \quad W_{x8}
\end{array} \]

\[ \begin{array}{c}
R_{x8} \quad R_{y8} \\
po \downarrow \quad po \downarrow \\
W_{y8} \quad W_{x8}
\end{array} \]
[OOTA-if example]
Solutions

[?]  

[?]  

[?]  

[Jeffrey and Riely, 2016]

[Kang et al., 2017]

[Pichon-Pharabod and Sewell, 2016]

[Podkopaev et al., 2016]
Solutions

[?]

[?]

[?]

[Jeffrey and Riely, 2016]

Promise MM, [Kang et al., 2017]

[Pichon-Pharabod and Sewell, 2016]

[Podkopaev et al., 2016]
Memory Models for PLs

<table>
<thead>
<tr>
<th></th>
<th>EC</th>
<th>CO</th>
<th>No OOTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC MM, [Lamport, 1979]</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Java MM, [Manson et al., 2005]</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>C/C++11 MM, [Batty et al., 2011]</td>
<td>✓</td>
<td>✓*</td>
<td>✗</td>
</tr>
<tr>
<td>Proposed solution [Kang et al., 2017]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Promise MM, for C/C++ and Java

Requirements:

- allow Efficient Compilation (x86, Power, ARM)
- validate Compiler Optimizations (merging, rearranging, etc)
- no Out-Of-Thin-Air
[Store Buffering in Promise]
Load Buffering in Promise

\[
\begin{align*}
  a & := [x]; & b & := [y]; \\
  [y] & := 1; & [x] & := 1;
\end{align*}
\]

Memory: \[\langle x : 0@0, \tau \rangle, \langle y : 0@0, \tau \rangle\]

Final values \(a = \_), \(b = \)_
Load Buffering in Promise

\[
\begin{align*}
  a & := [x]; & b & := [y]; \\
  [y] & := 1; & [x] & := 1;
\end{align*}
\]

Memory: \[\langle x : 0@0\tau \rangle, \langle y : 0@0\tau \rangle\]

Final values \(a = \_\), \(b = \_\)
Load Buffering in Promise

\[
\begin{align*}
\overrightarrow{\text{Promised}} \\
\begin{array}{c}
\begin{array}{c}
a \ := \ [x]; \\
[y] \ := \ 1;
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
b \ := \ [y]; \\
x \ := \ 1;
\end{array}
\end{array}
\end{align*}
\]

Memory: \[
[\langle x: 0@0_\tau \rangle, \langle y: 0@0_\tau \rangle, \\
\langle y: 1@1_\tau \rangle]
\]

Final values \( a = _\_, \ b = _\_ \)
Load Buffering in Promise

\[
\begin{align*}
a &:= [x]; \\
[y] &:= 1; \\
\text{Promised} &
\end{align*}
\]

\[
\begin{align*}
b &:= [y]; \\
[x] &:= 1;
\end{align*}
\]

Memory: \([\langle x : 0@0_\tau \rangle, \langle y : 0@0_\tau \rangle, \langle y : 1@1_\tau \rangle]\]

Final values \(a = \_, b = 1\)
Load Buffering in Promise

\[ a := [x]; \quad [y] := 1; \quad b := [y]; \quad [x] := 1; \]

Memory: \[ [\langle x : 0@0_\tau \rangle, \langle y : 0@0_\tau \rangle, \langle y : 1@1_\tau \rangle, \langle x : 1@1_\tau \rangle] \]

Final values \( a = \_ \), \( b = 1 \)
Load Buffering in Promise

\[
a := [x];
\]
\[
[y] := 1;
\]
\[
b := [y];
\]
\[
[x] := 1;
\]

Memory: \[
[\langle x : 0 \@ 0_\tau \rangle, \langle y : 0 \@ 0_\tau \rangle,
\langle y : 1 \@ 1_\tau \rangle, \langle x : 1 \@ 1_\tau \rangle]
\]

Final values \(a = 1, b = 1\)
Load Buffering in Promise

\[
\begin{align*}
  a & := [x]; & b & := [y]; \\
  [y] & := 1; & [x] & := 1;
\end{align*}
\]

Memory: \[
\left[ \langle x : 0\@0\tau \rangle, \langle y : 0\@0\tau \rangle, \langle y : 1\@1\tau \rangle, \langle x : 1\@1\tau \rangle \right]
\]

Final values \(a = 1, b = 1\)
ARM-Weak in Promise

\[ a := [x]; \quad b := [x]; \quad c := [y]; \]
\[ [x] := 1; \quad [y] := b; \quad [x] := c; \]

\[ a = 1? \]
ARM-Weak in Promise

\[
\begin{align*}
  & a := \langle x \rangle; & b := \langle x \rangle; & c := \langle y \rangle; \\
  & [x] := 1; & [y] := b; & [x] := c;
\end{align*}
\]

V1: \[\langle x:0@0_\tau, y:0@0_\tau \rangle\]  
V2: \[\langle x:0@0_\tau, y:0@0_\tau \rangle\]  
V3: \[\langle x:0@0_\tau, y:0@0_\tau \rangle\]

Memory: \[\langle x:0@0_\tau \rangle, \langle y:0@0_\tau \rangle\]

Values: \[a = \bot \quad b = \bot \quad c = \bot\]
ARM-Weak in Promise

\[ a := [x]; \quad b := [x]; \quad c := [y]; \]
\[ [x] := 1; \quad [y] := b; \quad [x] := c; \]

V1: \([x@0_\tau, y@0_\tau]\)  V2: \([x@0_\tau, y@0_\tau]\)  V3: \([x@0_\tau, y@0_\tau]\)

Memory: \([\langle x : 0@0_\tau \rangle, \langle y : 0@0_\tau \rangle, \langle x : 1@2_\tau \rangle]\]

Values: \(a = \bot\)  \(b = \bot\)  \(c = \bot\)
ARM-Weak in Promise

\[
\begin{align*}
& a := [x]; \\
& [x] := 1; \quad \text{Promised} \\
& b := [x]; \\
& [y] := b; \\
& c := [y]; \\
& [x] := c;
\end{align*}
\]

\[
\text{V1: } [x@0_\tau, y@0_\tau] \quad \text{V2: } [x@2_\tau, y@0_\tau] \quad \text{V3: } [x@0_\tau, y@0_\tau]
\]

\[
\text{Memory: } [\langle x : 0@0_\tau \rangle, \langle y : 0@0_\tau \rangle, \\
\langle x : 1@2_\tau \rangle]
\]

Values: \quad a = \bot \quad b = 1 \quad c = \bot
ARM-Weak in Promise

\[ a := [x]; \quad b := [x]; \quad c := [y]; \]

\[ [x] := 1; \quad [y] := b; \quad [x] := c; \]

\begin{align*}
V1: \; & [x@0_\tau, y@0_\tau] \\
V2: \; & [x@2_\tau, y@1_\tau] \\
V3: \; & [x@0_\tau, y@0_\tau]
\end{align*}

Memory:

\[ [\langle x : 0@0_\tau \rangle, \langle y : 0@0_\tau \rangle, \langle x : 1@2_\tau \rangle, \langle y : 1@1_\tau \rangle] \]

Values:

\[ a = \bot \quad b = 1 \quad c = \bot \]
ARM-Weak in Promise

\[
\begin{align*}
a & := [x]; \\
[x] & := 1; \\
\text{Promised} \\
b & := [x]; \\
[y] & := b; \\
c & := [y]; \\
[x] & := c;
\end{align*}
\]

V1: \([x@0_\tau, y@0_\tau]\)  V2: \([x@2_\tau, y@1_\tau]\)  V3: \([x@0_\tau, y@1_\tau]\)

Memory: \([\langle x : 0@0_\tau \rangle, \langle y : 0@0_\tau \rangle, \\
\langle x : 1@2_\tau \rangle, \langle y : 1@1_\tau \rangle]\)  

Values: \(a = \bot\) \hspace{1cm} b = 1 \hspace{1cm} c = 1
ARM-Weak in Promise

\[ a := [x]; \]
\[ [x] := 1; \]
\[ b := [x]; \]
\[ [y] := b; \]
\[ c := [y]; \]
\[ [x] := c; \]

V1: \([x@0_\tau, y@0_\tau]\]
V2: \([x@2_\tau, y@1_\tau]\]
V3: \([x@1_\tau, y@1_\tau]\]

Memory: \(\langle x : 0@0_\tau \rangle, \langle y : 0@0_\tau \rangle, \langle x : 1@2_\tau \rangle, \langle y : 1@1_\tau \rangle, \langle x : 1@1_\tau \rangle\)\]

Values: \(a = \bot\) \(b = 1\) \(c = 1\)
ARM-Weak in Promise

\[ a := [x]; \quad | \quad b := [x]; \quad | \quad c := [y]; \]

\[ [x] := 1; \quad | \quad [y] := b; \quad | \quad [x] := c; \]

V1: \([x@1_\tau, y@0_\tau]\) \quad V2: \([x@2_\tau, y@1_\tau]\) \quad V3: \([x@1_\tau, y@1_\tau]\)

Memory: \([\langle x : 0@0_\tau \rangle, \langle y : 0@0_\tau \rangle, \langle x : 1@2_\tau \rangle, \langle y : 1@1_\tau \rangle, \langle x : 1@1_\tau \rangle] \]

Values: \quad a = 1 \quad b = 1 \quad c = 1
ARM-Weak in Promise

\[
a : = \left[ x \right]; \quad b : = \left[ x \right]; \quad c : = \left[ y \right];
\]
\[
\left[ x \right] : = 1; \quad \left[ y \right] : = b; \quad \left[ x \right] : = c;
\]

V1: \([x@2_\tau, y@0_\tau]\)  V2: \([x@2_\tau, y@1_\tau]\)  V3: \([x@1_\tau, y@1_\tau]\)

Memory: \[
\langle x : 0@0_\tau \rangle, \langle y : 0@0_\tau \rangle, \\
\langle x : 1@2_\tau \rangle, \langle y : 1@1_\tau \rangle, \\
\langle x : 1@1_\tau \rangle
\]

Values: \[
a = 1 \quad b = 1 \quad c = 1
\]
Agenda

- Memory models. What, why, and when to care

- CPU and PL MMs. Promise MM

- Compilation correctness for Promise MM to \{x86, Power, ARM\}
Compilation Correctness

Source language

S
Compilation Correctness

\[ S \rightarrow T \]

Target language
Compilation Correctness

\[ \text{compile} : S \rightarrow T \]
Compilation Correctness

\[\text{compile} : S \rightarrow T\]
Compilation Correctness

\[
\text{compile} : S \rightarrow T
\]
Compilation Correctness

\[
\text{compile} : S \rightarrow T
\]

\[
\forall \text{Prog} \in S.
\]

\[
\left[ \text{compile}(\text{Prog}) \right]_T \subseteq \left[ \text{Prog} \right]_S.
\]
Compilation Targets

- **x86-TSO**, [Owens et al., 2009]
- **Power**, [Alglave et al., 2014]
- **ARMv8 POP**, [Flur et al., 2016]
- **ARMv8.3**, [Pulte et al., 2018]
Compilation to x86-TSO
Compilation to x86-TSO

- $x86$-TSO = SC + transformations
  
  [Lahav and Vafeiadis, 2016]
Compilation to x86-TSO

- x86-TSO = SC + transformations
  [Lahav and Vafeiadis, 2016]
  - Reordering of independent write-read
  - Read-after-write elimination
Compilation to x86-TSO

- x86-TSO = SC + transformations
  [Lahav and Vafeiadis, 2016]
  - Reordering of independent write-read
  - Read-after-write elimination

- Transformations are sound in Promise
  [Kang et al., 2017]
Compilation to x86-TSO

- \(x86\text{-TSO} = \text{SC} + \text{transformations}\)
  - [Lahav and Vafeiadis, 2016]
  - Reordering of independent write-read
  - Read-after-write elimination

- Transformations are sound in Promise
  - [Kang et al., 2017]

- \(\text{SC} \subset \text{Promise}\)
Compilation to Power

• Power = StrongPower + transformation [Lahav and Vafeiadis, 2016]
• Reordering of independent instructions
• Transformation is sound in Promise [Kanget al., 2017]
• StrongPower ⊆ promise-free version of Promise
Compilation to Power

- $\text{Power} = \text{StrongPower} + \text{transformation}$
  
  [Lahav and Vafeiadis, 2016]
Compilation to Power

- **Power = StrongPower + transformation**
  - [Lahav and Vafeiadis, 2016]
  - Reordering of independent instructions
Compilation to Power

- \textbf{Power} = \textbf{StrongPower} + \textbf{transformation} \\
  \quad [\text{Lahav and Vafeiadis, 2016}]
  - Reordering of independent instructions

- Transformation is sound in Promise \\
  \quad [\text{Kang et al., 2017}]
Compilation to Power

- **Power = StrongPower + transformation**
  
  - Reordering of independent instructions

- **Transformation is sound in Promise**

  [Kang et al., 2017]

- **StrongPower ⊆ promise-free version of Promise**

  [Lahav and Vafeiadis, 2016]
Scheme isn’t applicable to ARM POP
Counterexample. ARM-Weak

\[ a := [x]; // 1 \quad b := [x]; \quad c := [y]; \]
\[ [x] := 1; \quad [y] := b; \quad [x] := c; \]
Counterexample. ARM-Weak

\[ a := [x]; \quad // 1 \quad b := [x]; \quad c := [y]; \]
\[ [x] := 1; \quad [y] := b; \quad [x] := c; \]

**Allowed in ARM POP**
Counterexample. ARM-Weak

\[ a := [x]; \quad \text{// 1} \quad b := [x]; \quad c := [y]; \]

\[ [x] := 1; \quad [y] := b; \quad [x] := c; \]

**Allowed** in ARM POP;

**Cannot** be explained by transformations over a stronger model
Main Differences between Promise and ARMv8 POP

1. Promise can execute only writes out-of-order

2. ARMv8 POP doesn’t totally order writes to a specific location
Main Proof Ingredients

1. “Lagging” simulation

2. ARMv8 POP + timestamps
Main Proof Ingredients

1. “Lagging” simulation

2. ARMv8 POP + timestamps
“Lagging” Simulation

\[
a := [x];
b := [y];
c := [x];
[y] := 1;
\]

...
“Lagging” Simulation

\[
\begin{align*}
a &:= [x]; \\
b &:= [y]; \\
c &:= [x]; \\
[y] &:= 1;
\end{align*}
\]
“Lagging” Simulation

\[
\begin{align*}
  a & := [x]; \\
  b & := [y]; \\
  c & := [x]; \\
  [y] & := 1;
\end{align*}
\]
“Lagging” Simulation

\[ a := \llbracket x \rrbracket; \]
\[ b := \llbracket y \rrbracket; \]
\[ c := \llbracket x \rrbracket; \]
\[ \llbracket y \rrbracket := 1; \]

FULLY executed by ARM
“Lagging” Simulation

- $a := [x]$; Fully executed by ARM
- $b := [y]$;
- $c := [x]$; Partially executed by ARM
- $[y] := 1$;
“Lagging” Simulation

\[ a := [x]; \quad \text{Fully executed by ARM} \]
\[ b := [y]; \]
\[ c := [x]; \quad \text{Partially executed by ARM} \]
\[ [y] := 1; \quad \text{Not executed by ARM} \]
“Lagging” Simulation

\begin{align*}
a &:= [x]; \\
b &:= [y]; \\
c &:= [x]; \\
[y] &:= 1;
\end{align*}
“Lagging” Simulation

\[ I \rightarrow \text{simulation relation} \]

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


“Lagging” Simulation

\[
\begin{align*}
\text{Promise} & \quad \textcolor{green}{a := [x];} \\
& \quad \textcolor{gray}{b := [y];} \\
& \quad \textcolor{yellow}{c := [x];} \\
& \quad [y] := 1;
\end{align*}
\]

\[
\mathcal{I} = \mathcal{I}_{\text{Promise is waiting}} \cup \mathcal{I}_{\text{Promise is executing}}
\]
“Lagging” Simulation

\[ I = I_{\text{Promise is waiting}} \cup I_{\text{Promise is executing}} \]
“Lagging” Simulation

\[ I = I_{\text{Promise is waiting}} \cup I_{\text{Promise is executing}} \]

\[
\begin{align*}
  &a := [x]; \\
  &b := [y]; \\
  &c := [x]; \\
  &\quad [y] := 1;
\end{align*}
\]
“Lagging” Simulation

\[ I = I_{\text{Promise is waiting}} \cup I_{\text{Promise is executing}} \]

- \[ a := [x]; \]
- \[ b := [y]; \]
- \[ c := [x]; \]
- \[ y := 1; \]

Promise models.
“Lagging” Simulation

\[ a := [x]; \]
\[ b := [y]; \]
\[ c := [x]; \]
\[ [y] := 1; \]

\[ I = I_{\text{Promise is waiting}} \cup I_{\text{Promise is executing}} \]
“Lagging” Simulation

\[ a := [x]; \]
\[ b := [y]; \]
\[ c := [x]; \]
\[ [y] := 1; \]

\[ I = I_{\text{Promise is waiting}} \cup I_{\text{Promise is executing}} \]
“Lagging” Simulation

\[ a := [x]; \]
\[ b := [y]; \]
\[ c := [x]; \]
\[ [y] := 1; \]

\[ \mathcal{I} = \mathcal{I}_{\text{Promise is waiting}} \cup \mathcal{I}_{\text{Promise is executing}} \]
“Lagging” Simulation

\[a := [x];\]
\[b := [y];\]
\[c := [x];\]
\[[y] := 1;\]

\[\mathcal{I} = \mathcal{I}_{\text{Promise is waiting}} \cup \mathcal{I}_{\text{Promise is executing}}\]
“Lagging” Simulation

\[ a := [x]; \]
\[ b := [y]; \]
\[ c := [x]; \]
\[ [y] := 1; \]

\[ I = I_{\text{Promise is waiting}} \cup I_{\text{Promise is executing}} \]
“Lagging” Simulation

\[ a := [x]; \]
\[ b := [y]; \]
\[ c := [x]; \]
\[ [y] := 1; \]

\[ \mathcal{I} = \mathcal{I}_{\text{Promise is waiting}} \cup \mathcal{I}_{\text{Promise is executing}} \]
Main Proof Ingredients

1. “Lagging” simulation

2. ARMv8 POP + timestamps
ARMv8 POP

\[ [x] := 1; \parallel [x] := 2; \]
ARMv8 POP

\[
[x] := 1; \quad \|
\]

\[
[x] := 2;
\]
ARMv8 POP

\[
\begin{align*}
[x] & := 1; \\
[x] & := 2; \\
[x] & \leftarrow 1
\end{align*}
\]
ARMv8 POP

\[ [x] := 1; \]

\[ [x] \leftarrow 1 \]

\[ [x] := 2; \]

\[ [x] \leftarrow 2 \]
ARMv8 POP

\[ [x] := 1; \quad \parallel \quad [x] := 2; \]

\[ [x] \leftarrow 1 \quad \parallel \quad [x] \leftarrow 2 \]
ARMv8 POP

\[ [x] := 1; \quad [x] := 2; \]

\[ [x] \leftarrow 2 \]

\[ [x] \leftarrow 1 \]
ARMv8 POP

\[
\begin{align*}
[x] & := 1; \\
[x] & := 2;
\end{align*}
\]

May propagate from left
Cannot propagate from right!
ARMv8 POP

\[
\begin{align*}
[x] & := 1; \\
\tau & := 1 \\
[x] & := 2; \\
\tau & := 2 \\
[x] & := 1 \\
\tau & := 2 \\
[x] & := 2 \\
\end{align*}
\]

May propagate from left
Cannot propagate from right!
ARMv8 POP

\[
\begin{align*}
[x] & := 1; \\
[x] & := 2;
\end{align*}
\]

May propagate from left

Cannot propagate from right!
ARMv8 POP

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!

Let’s determine the order beforehand!
ARMv8 POP + Timestamps

\[ x := 1; \] \quad \text{and} \quad \[ x := 2; \]
ARMv8 POP + Timestamps

\[ x \leftarrow 1 @ 3_T \]

\[ x := 1; \]
\[ x := 2; \]
ARMv8 POP + Timestamps

\[
\begin{align*}
[x] &:= 1; \\
[x] &:= 2;
\end{align*}
\]

\[
\begin{align*}
[x] := 1 @ 3_T &\quad & [x] := 2 @ 8_T
\end{align*}
\]
ARMv8 POP + Timestamps

May propagate from left
ARMv8 POP + Timestamps

[x] := 1;

[x] ← 1 @ 3τ

[x] := 2;

[x] ← 2 @ 8τ

Cannot propagate from right!
Proof Structure

1. Introduce ARM+τ

2. Prove equivalence between ARM+τ and ARMv8 POP

3. Show “lagging” simulation from Promise to ARM+τ
How to prove correctness of compilation?
How to prove correctness of compilation?

Standard technique:
Simulation
Simulation works for operational semantics
Simulation works for operational semantics

How to simulate graphs?
Simulation works for operational semantics

How to **simulate graphs?**

Traverse in proper order!
Traverse of ARMv8.3 execution

\[
a := [x]; \quad b := [y];
\]
\[
[y] := 1; \quad [x] := 1;
\]
Traverse of ARMv8.3 execution

\[
\begin{align*}
  a & := [x]; & b & := [y]; \\
  [y] & := 1; & [x] & := 1;
\end{align*}
\]
Traverse of ARMv8.3 execution

\[
\begin{align*}
  a & := [x]; & b & := [y]; \\
  [y] & := 1; & [x] & := 1;
\end{align*}
\]
Traverse of ARMv8.3 execution

\[ a := [x]; \quad | \quad b := [y]; \]
\[ [y] := 1; \quad | \quad [x] := 1; \]
Traverse of ARMv8.3 execution

\[
a := [x]; \quad \| \quad b := [y];
\]
\[
[y] := 1; \quad \| \quad [x] := 1;
\]
Traverse of ARMv8.3 execution

\[ a := [x]; \quad | \quad b := [y]; \]
\[ [y] := 1; \quad | \quad [x] := 1; \]
Traverse of ARMv8.3 execution

\[
a := [x]; \quad \text{or} \quad b := [y];\\
[y] := 1; \quad \text{or} \quad [x] := 1;
\]

Diagram:

- Rx1
- Ry1
- Wy1
- Wx1

- Covered
- Issued
Traverse of ARMv8.3 execution

\[ a := \{x\}; \quad b := \{y\}; \]
\[ \{y\} := 1; \quad \{x\} := 1; \]
Traversal formally

Cover step:

\[ G \vdash \langle C, I \rangle \rightarrow \langle C \cup \{a\}, I \rangle \]

Issue step:

\[ G \vdash \langle C, I \rangle \rightarrow \langle C, I \cup \{w\} \rangle \]
Traversal formally

Cover step:

\[ G \vdash \langle C, I \rangle \rightarrow \langle C \cup \{a\}, I \rangle \]

Issue step:

\[ G \vdash \langle C, I \rangle \rightarrow \langle C, I \cup \{w\} \rangle \]
Proof Structure

1. Prove Promise simulates traversal
Proof Structure

1. Prove Promise simulates traversal

2. Show completeness of traversal
Proof Structure

1. Prove Promise simulates traversal

2. Show completeness of traversal
   \[ \forall G. \ G \in \text{Consistent}(\text{ARMv8.3}) \Rightarrow G \vdash \langle \emptyset, \emptyset \rangle \rightarrow^* \langle G.\text{Events}, G.\text{Writes} \rangle \]
Takeaway

- MMs are important and complicated, but locks help
- Problems in existing MMs, but there are solutions
- Not all MMs might be explained by reorderings
Takeaway

• MMs are important and complicated, but locks help

• Problems in existing MMs, but there are solutions

• Not all MMs might be explained by reorderings

http://podkopaev.net

Thank you!
Links I


Links II

- **Lamport, L. (1979).**
  How to make a multiprocessor computer that correctly executes multiprocess programs.

- **Manson, J., Pugh, W., and Adve, S. V. (2005).**
  The Java memory model.
  In *POPL 2005*, pages 378–391. ACM.

  A better x86 memory model: x86-TSO.

- **Pichon-Pharabod, J. and Sewell, P. (2016).**
  A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions.
  In *POPL 2016*, pages 622–633. ACM.

- **Podkopaev, A., Sergey, I., and Nanevski, A. (2016).**
  Operational aspects of C/C++ concurrency.
  *CoRR*, abs/1606.01400.
Links III