

# Weak Memory

Anton Podkopaev

St. Petersburg University, JetBrains Research, Russia

13.12.2017

# 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**}

# 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 model (MM)** is  
concurrent system's  
semantics



# Sequential consistency [Lamport, 1979]

# Sequential consistency [Lamport, 1979]

*system's behaviors —  
program interleavings*

# Execution in SC

$$\begin{array}{l} \xrightarrow{\quad} \\ [x] := 1; \\ a := [y]; \end{array} \parallel \begin{array}{l} \xrightarrow{\quad} \\ [y] := 1; \\ b := [x]; \end{array}$$

Values

$a = \perp; b = \perp$

Memory

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

# Execution in SC

$$\begin{array}{l} [x] := 1; \\ \xrightarrow{\quad} a := [y]; \end{array} \parallel \begin{array}{l} \xrightarrow{\quad} [y] := 1; \\ b := [x]; \end{array}$$

Values

$a = \perp; b = \perp$

Memory

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

# Execution in SC

$$\begin{array}{l} [x] := 1; \\ \xrightarrow{\quad} \\ a := [y]; \end{array} \parallel \begin{array}{l} [y] := 1; \\ \xrightarrow{\quad} \\ b := [x]; \end{array}$$

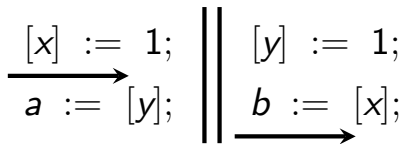
Values

$a = \perp; b = \perp$

Memory

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

# Execution in SC



Values

$a = \perp; b = 1$

Memory

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

# Execution in SC

$$\begin{array}{l|l} [x] := 1; & [y] := 1; \\ a := [y]; & b := [x]; \\ \hline \longrightarrow & \longrightarrow \end{array}$$

Values

$a = 1; b = 1$

Memory

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





Is it the same in reality?

Is it the same in reality?  
Let's check!



# 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
- . . .



Correct

# 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
- ...

} Correct for **one** thread

Lead to strange concurrent behaviors



# 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 *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}

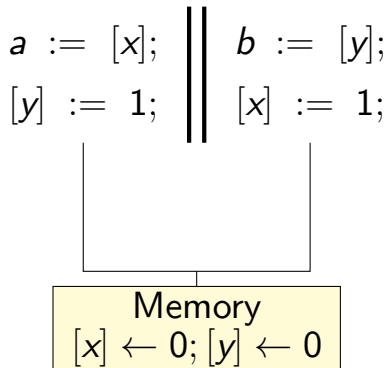
[Store Buffering in  
x86-TSO  
[Owens et al., 2009]]

# Load Buffering

$$\begin{array}{l} a := [x]; \\ [y] := 1; \end{array} \parallel \begin{array}{l} b := [y]; \\ [x] := 1; \end{array}$$

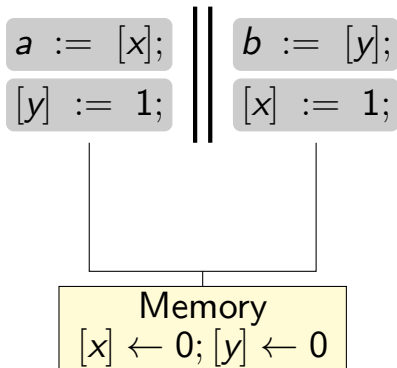
Final values  $a = 1$ ,  $b = 1$

# Load Buffering in ARMv8 POP



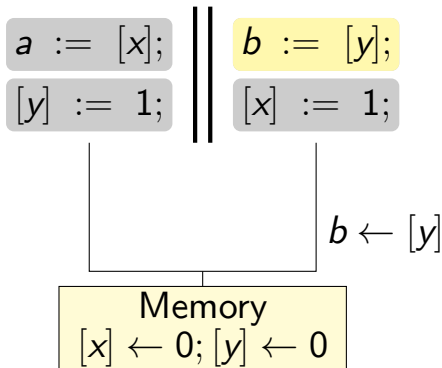
Final values  $a = \_$ ,  $b = \_$

# Load Buffering in ARMv8 POP



Final values  $a = \_$ ,  $b = \_$

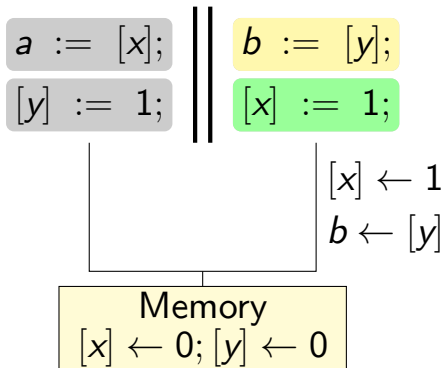
# Load Buffering in ARMv8 POP



Final values  $a = \_$ ,  $b = \_$

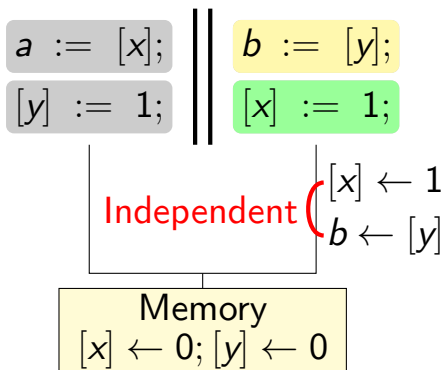


# Load Buffering in ARMv8 POP



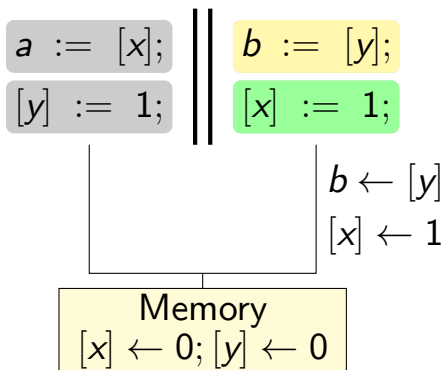
Final values  $a = \_$ ,  $b = \_$

# Load Buffering in ARMv8 POP



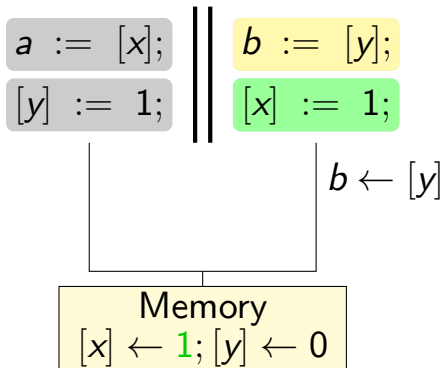
Final values  $a = \_$ ,  $b = \_$

# Load Buffering in ARMv8 POP



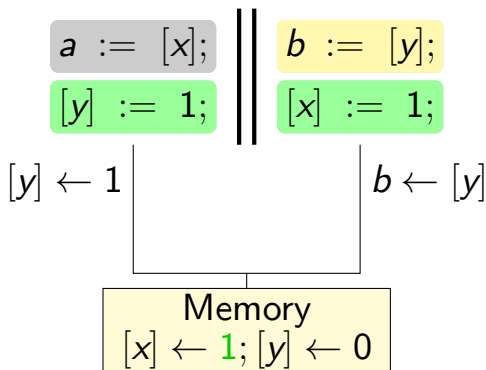
Final values  $a = \_$ ,  $b = \_$

# Load Buffering in ARMv8 POP



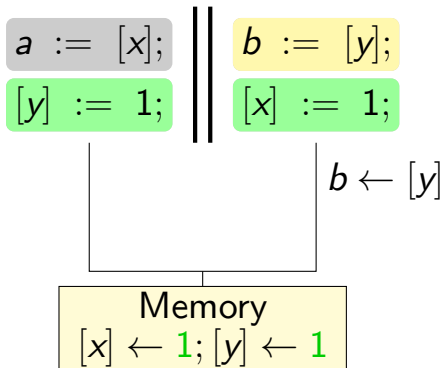
Final values  $a = \_$ ,  $b = \_$

# Load Buffering in ARMv8 POP



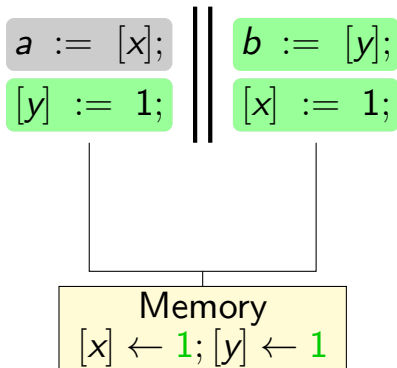
Final values  $a = \_$ ,  $b = \_$

# Load Buffering in ARMv8 POP



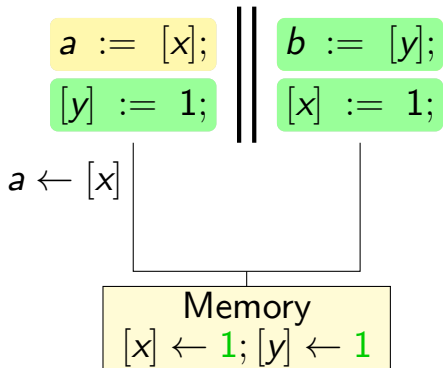
Final values  $a = \_$ ,  $b = \_$

# Load Buffering in ARMv8 POP



Final values  $a = \_$ ,  $b = 1$

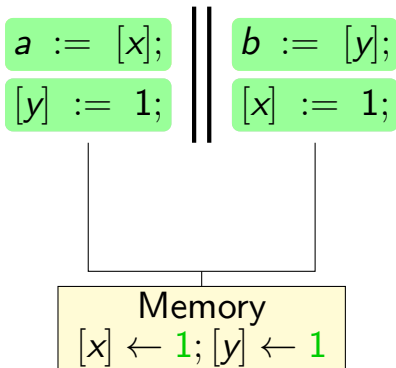
# Load Buffering in ARMv8 POP



Final values  $a = \_$ ,  $b = 1$



# Load Buffering in ARMv8 POP



Final values  $a = 1$ ,  $b = 1$

# ARM-Weak in ARMv8 POP

$$\begin{array}{l} a := [x]; \\ [x] := 1; \end{array} \parallel \begin{array}{l} b := [x]; \\ [y] := b; \end{array} \parallel \begin{array}{l} c := [y]; \\ [x] := c; \end{array}$$

$a = 1?$

# ARM-Weak in ARMv8 POP

$$\begin{array}{l}
 a := [x]; \quad || \quad b := [x]; \quad || \quad c := [y]; \\
 [x] := 1; \quad || \quad [y] := b; \quad || \quad [x] := c;
 \end{array}$$

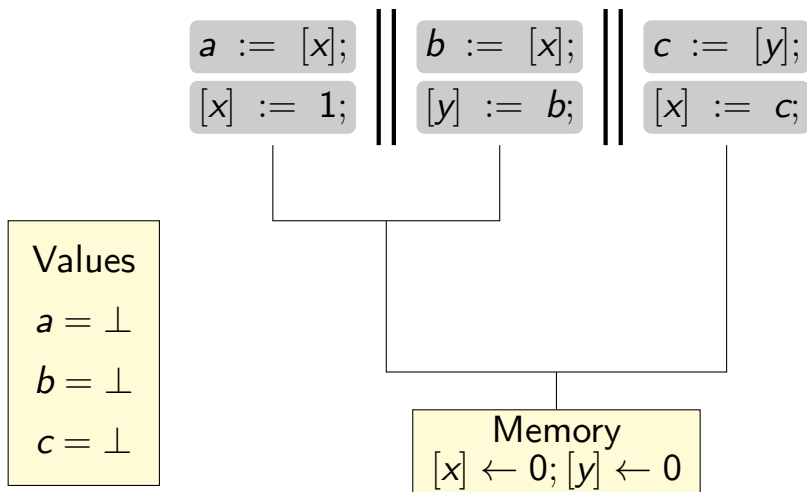
Values

 $a = \perp$ 
 $b = \perp$ 
 $c = \perp$ 

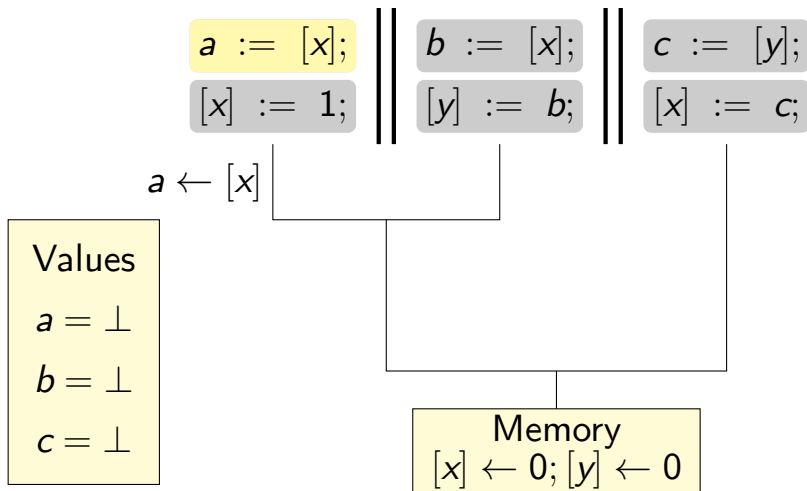
Memory

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

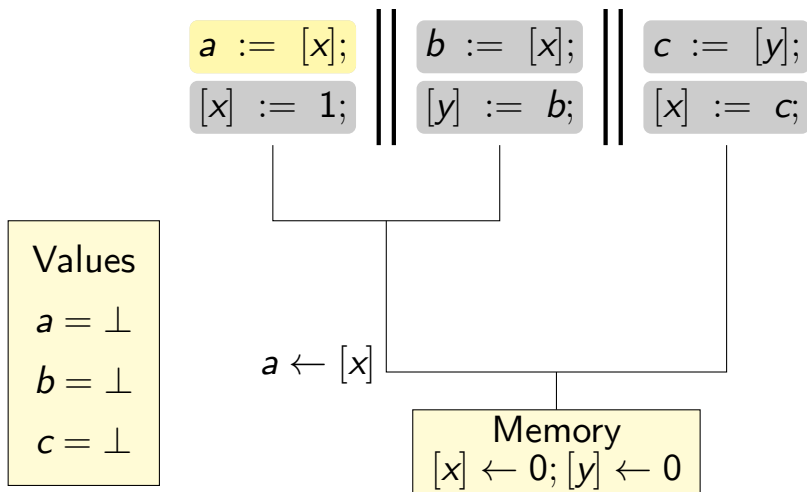
# ARM-Weak in ARMv8 POP



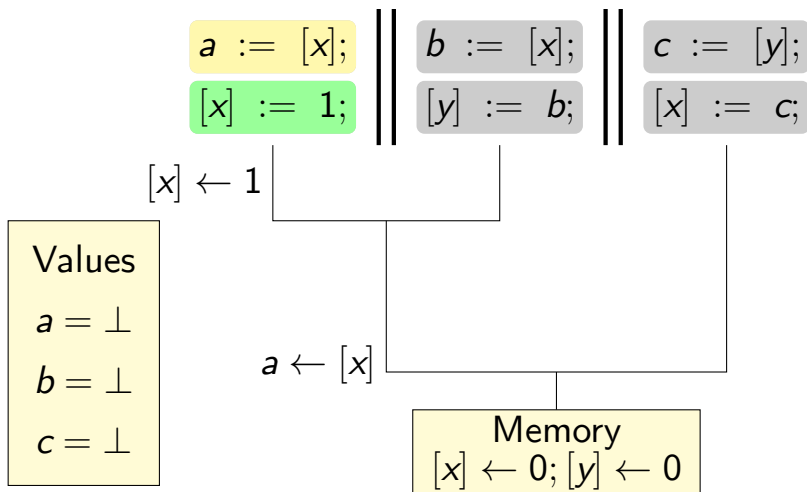
# ARM-Weak in ARMv8 POP



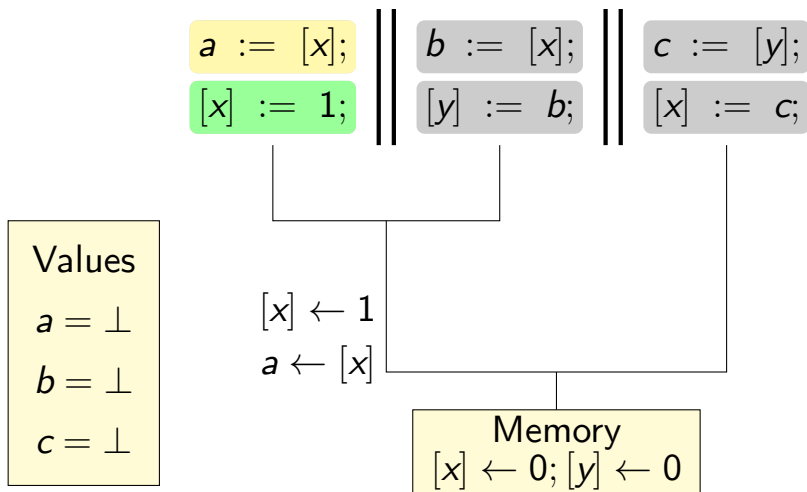
# ARM-Weak in ARMv8 POP



# ARM-Weak in ARMv8 POP

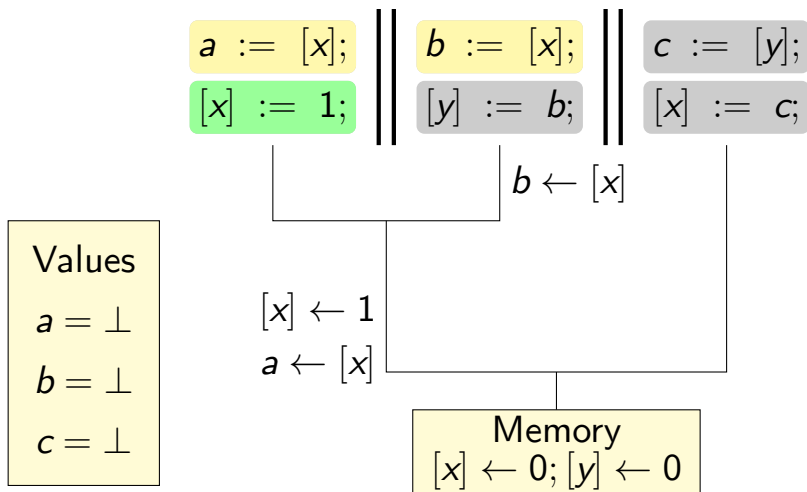


# ARM-Weak in ARMv8 POP

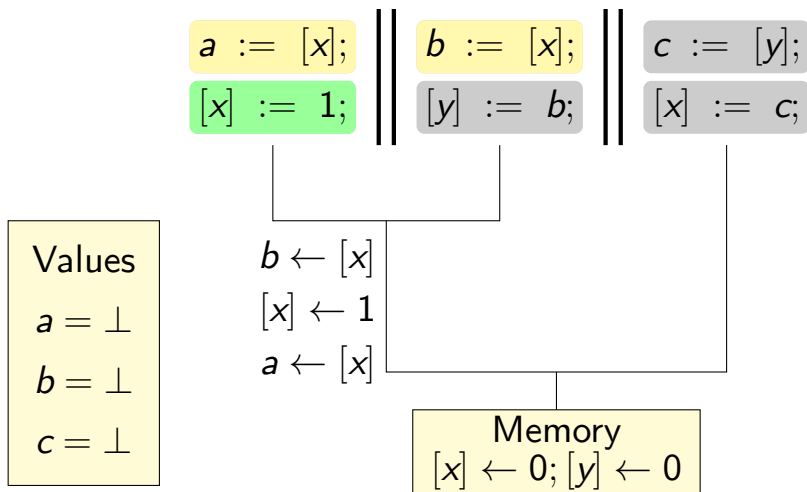




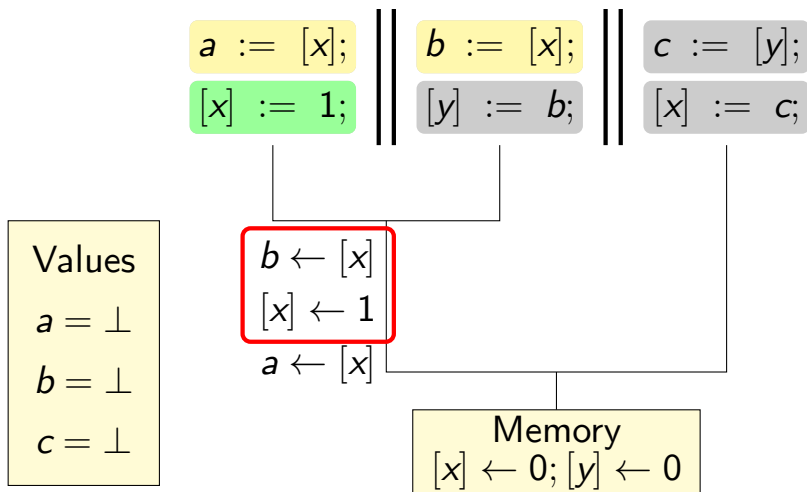
# ARM-Weak in ARMv8 POP



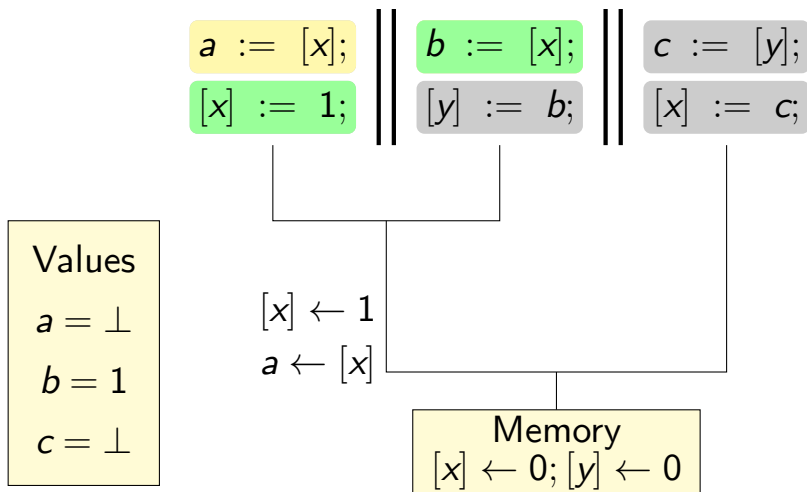
# ARM-Weak in ARMv8 POP



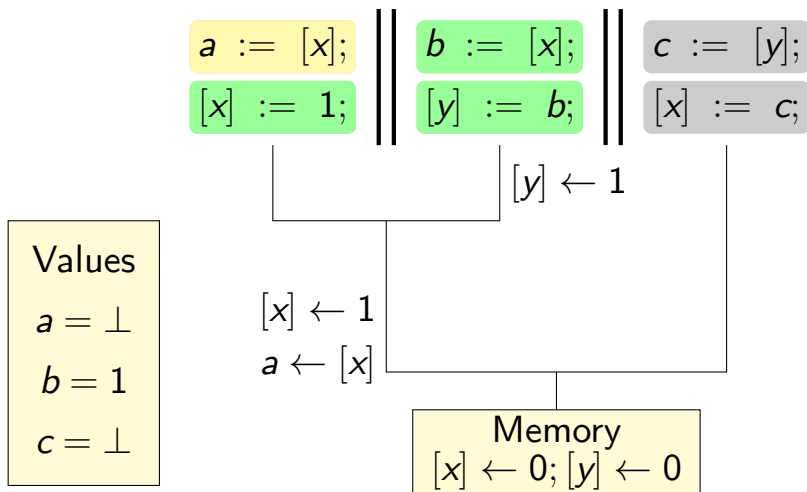
# ARM-Weak in ARMv8 POP



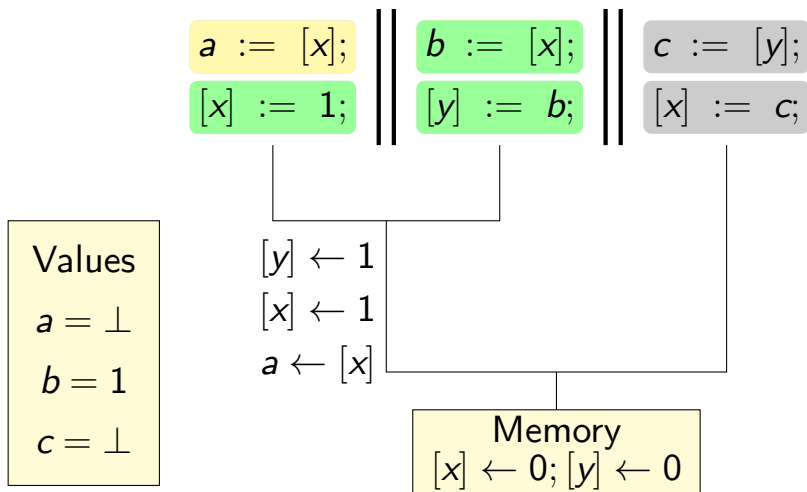
# ARM-Weak in ARMv8 POP



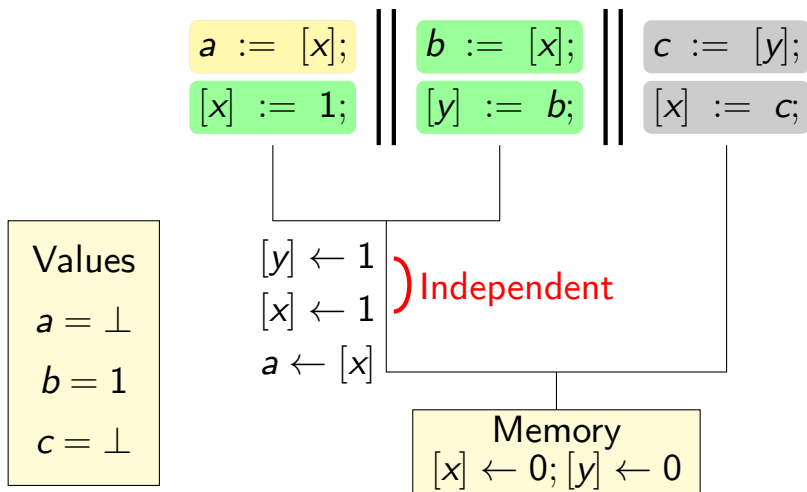
# ARM-Weak in ARMv8 POP



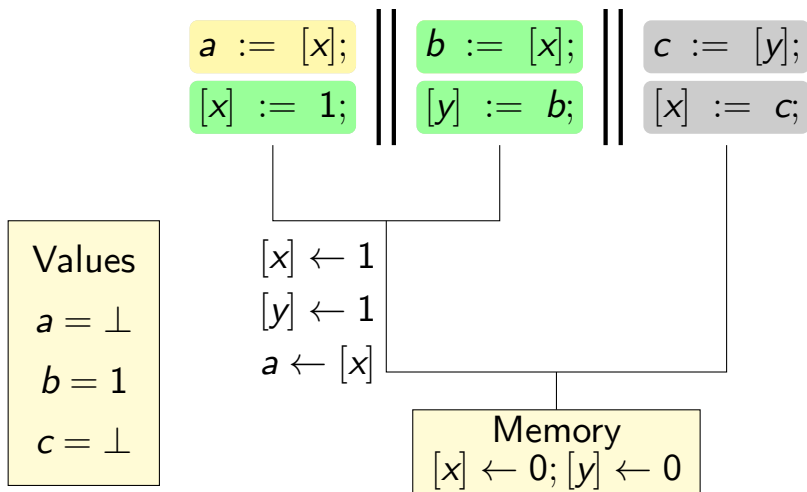
# ARM-Weak in ARMv8 POP



# ARM-Weak in ARMv8 POP

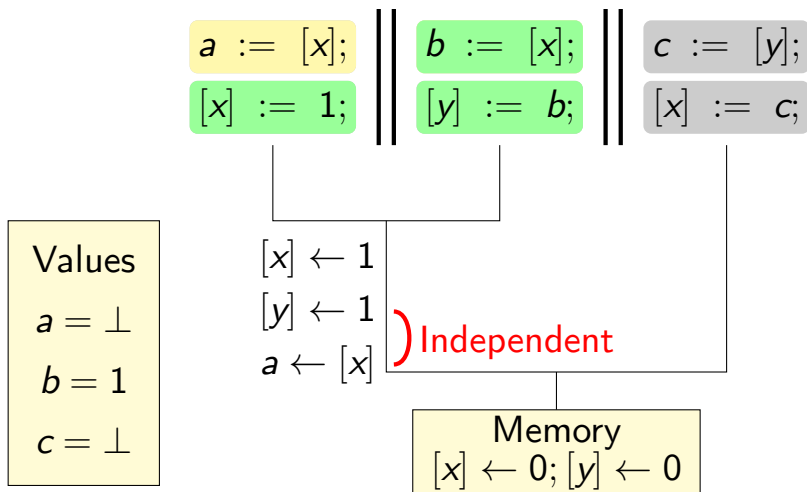


# ARM-Weak in ARMv8 POP

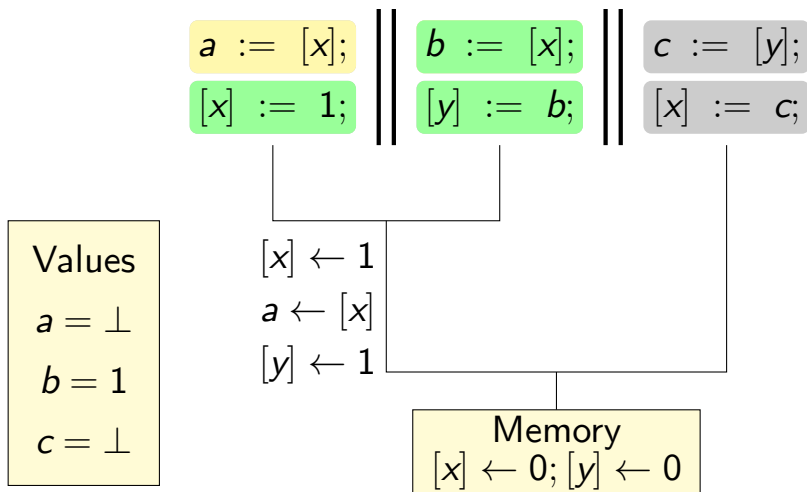




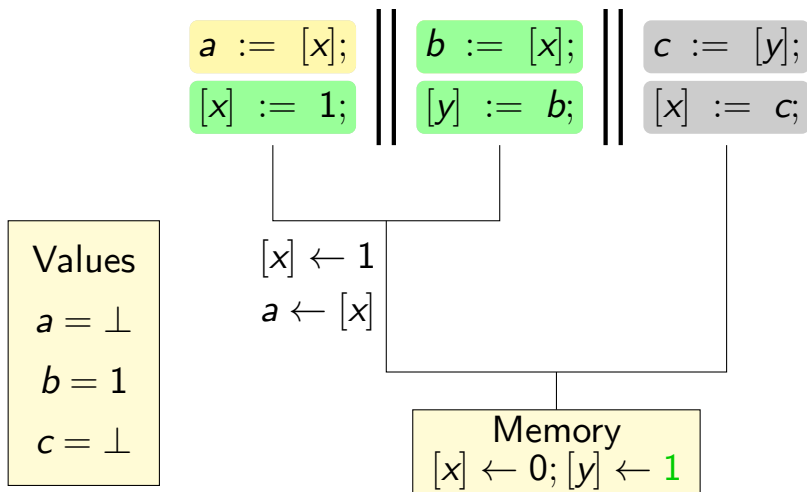
# ARM-Weak in ARMv8 POP



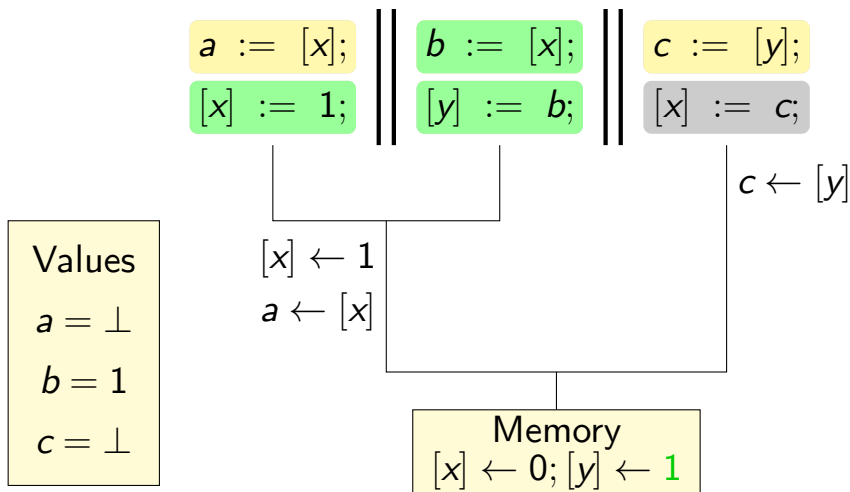
# ARM-Weak in ARMv8 POP



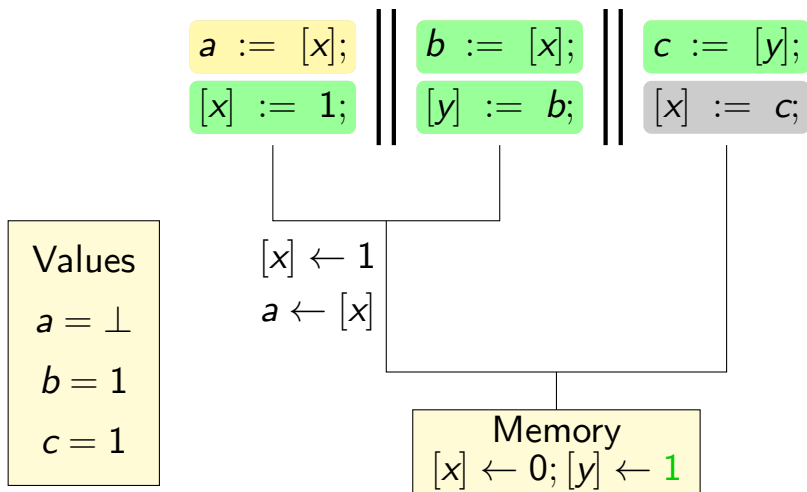
# ARM-Weak in ARMv8 POP



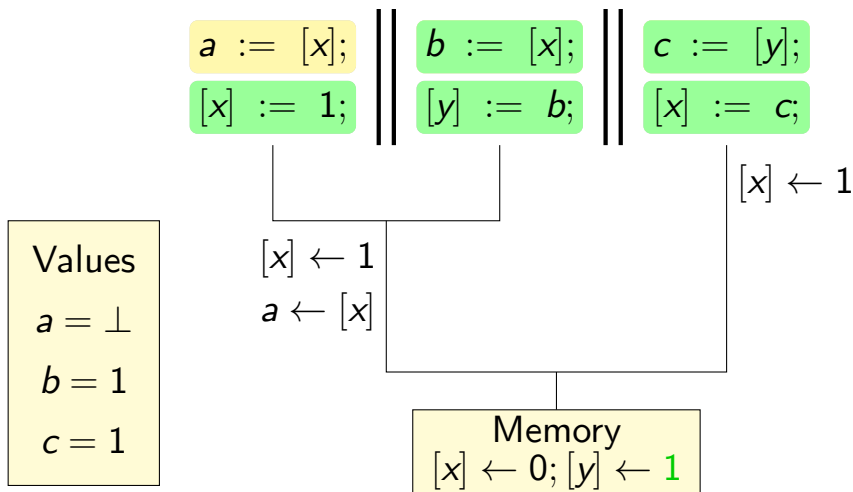
# ARM-Weak in ARMv8 POP



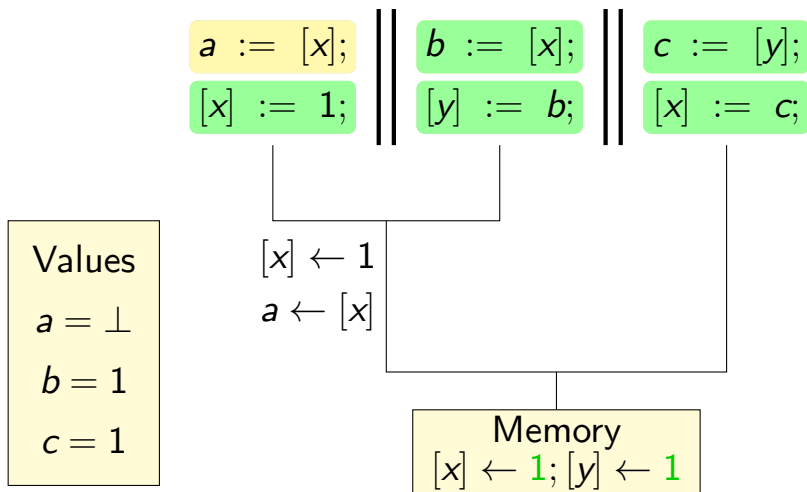
# ARM-Weak in ARMv8 POP



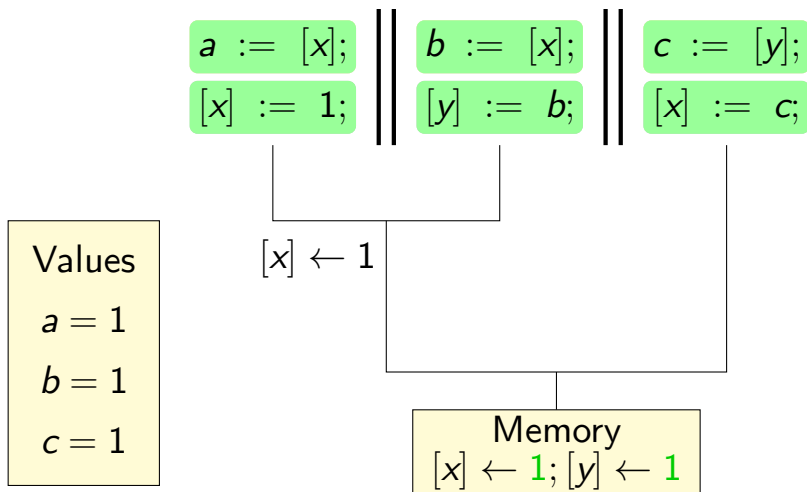
# ARM-Weak in ARMv8 POP



# ARM-Weak in ARMv8 POP



# ARM-Weak in ARMv8 POP





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

$$\begin{array}{l} [x] := 1; \\ a := [y]; \end{array} \parallel \begin{array}{l} [y] := 1; \\ b := [x]; \end{array}$$



# 1. Efficient Compilation

Source (SC MM)

$$\begin{array}{l} [x] := 1; \\ a := [y]; \end{array} \parallel \begin{array}{l} [y] := 1; \\ b := [x]; \end{array}$$

# 1. Efficient Compilation

Source (SC MM)

$$\begin{array}{l} [x] := 1; \\ a := [y]; \end{array} \parallel \begin{array}{l} [y] := 1; \\ b := [x]; \end{array}$$

Target (x86 MM)

# 1. Efficient Compilation

Source (SC MM)

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



Target (x86 MM)

$[x] := 1;$		$[y] := 1;$
mfence;		mfence;
$a := [y];$		$b := [x];$

# 1. Efficient Compilation

Source (SC MM)

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

**Not efficient**

Target (x86 MM)

$[x] := 1;$		$[y] := 1;$
mfence;		mfence;
$a := [y];$		$b := [x];$

## 2. Compiler Optimizations

## 2. Compiler Optimizations

$$\begin{array}{l} [x] := 1; \\ a := [y]; \end{array} \parallel \begin{array}{l} [y] := 1; \\ b := [x]; \end{array}$$

## 2. Compiler Optimizations

Original

$$\begin{array}{l} [x] := 1; \\ a := [y]; \end{array} \parallel \begin{array}{l} [y] := 1; \\ b := [x]; \end{array}$$

## 2. Compiler Optimizations

Original

$$\begin{array}{l} [x] := 1; \\ a := [y]; \end{array} \parallel \parallel \begin{array}{l} [y] := 1; \\ b := [x]; \end{array}$$

Optimized

$$\begin{array}{l} a := [y]; \\ [x] := 1; \end{array} \parallel \parallel \begin{array}{l} [y] := 1; \\ b := [x]; \end{array}$$



## 2. Compiler Optimizations

Original

$$\left[ \begin{array}{l} [x] := 1; \\ a := [y]; \end{array} \parallel \begin{array}{l} [y] := 1; \\ b := [x]; \end{array} \right]$$

UI

Optimized

$$\left[ \begin{array}{l} a := [y]; \\ [x] := 1; \end{array} \parallel \begin{array}{l} [y] := 1; \\ b := [x]; \end{array} \right]$$

# 3. No Out-Of-Thin-Air

### 3. No Out-Of-Thin-Air

$$\begin{array}{l} a := [x]; \\ [y] := a; \end{array} \parallel \begin{array}{l} b := [y]; \\ [x] := b; \end{array}$$

### 3. No Out-Of-Thin-Air

$$\begin{array}{l} a := [x]; \\ [y] := a; \end{array} \parallel \begin{array}{l} b := [y]; \\ [x] := b; \end{array}$$

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

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

## Requirements:

- allow **E**fficient **C**ompilation (x86, Power, ARM)
- validate **C**ompiler **O**ptimizations (merging, rearranging, etc)
- no **O**ut-**O**f-**T**hin-**A**ir

# Memory Models for PLs

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

## Requirements:

- allow **E**fficient **C**ompilation (x86, Power, ARM)
- validate **C**ompiler **O**ptimizations (merging, rearranging, etc)
- no **O**ut-**O**f-**T**hin-**A**ir



# Memory Models for PLs

	EC	CO	No OOTA
SC MM, [Lamport, 1979]	✗	✗	✓
Java MM, [Manson et al., 2005]	✓	✗	✓
C/C++11 MM, [Batty et al., 2011]	✓	✓*	✗

## Requirements:

- allow **E**fficient **C**ompilation (x86, Power, ARM)
- validate **C**ompiler **O**ptimizations (merging, rearranging, etc)
- no **O**ut-**O**f-**T**hin-**A**ir

# Memory Models for PLs

	EC	CO	No OOTA
SC MM, [Lamport, 1979]	✗	✗	✓
Java MM, [Manson et al., 2005]	✓	✗	✓
C/C++11 MM, [Batty et al., 2011]	✓	✓*	✗

## Requirements:

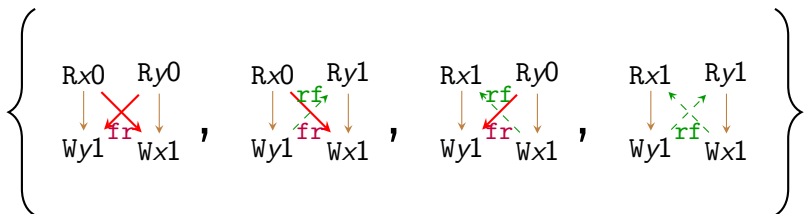
- allow **E**fficient **C**ompilation (x86, Power, ARM)
- validate **C**ompiler **O**ptimizations (merging, rearranging, etc)
- no **O**ut-**O**f-**T**hin-**A**ir

C/C++11 MM has OOTA

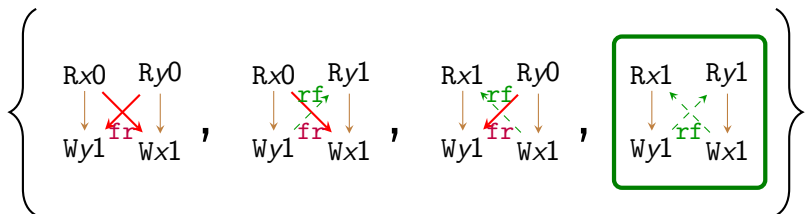
C/C++11 MM has OOTA.  
Why?

C/C++11 MM is  
**axiomatic** MM

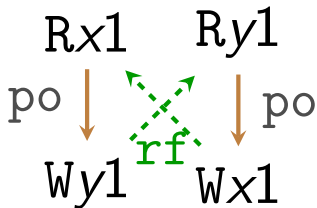
# Load Buffering in C/C++11

$$\begin{array}{l|l}
 a := [x]; & b := [y]; \\
 [y] := 1; & [x] := 1;
 \end{array}$$


# Load Buffering in C/C++11

$$\begin{array}{l|l}
 a := [x]; & b := [y]; \\
 [y] := 1; & [x] := 1;
 \end{array}$$


# Load Buffering in C/C++11

$$\begin{array}{l}
 a := [x]; \\
 [y] := 1;
 \end{array}
 \parallel
 \parallel
 \begin{array}{l}
 b := [y]; \\
 [x] := 1;
 \end{array}$$




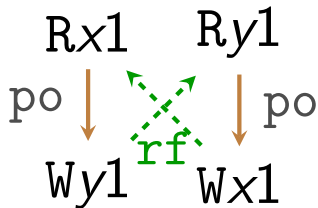
# Load Buffering in C/C++11

$$\begin{array}{l}
 a := [x]; \\
 [y] := 1;
 \end{array}
 \parallel
 \begin{array}{l}
 b := [y]; \\
 [x] := 1;
 \end{array}$$

Axioms:

1. **hb** is acyclic

...



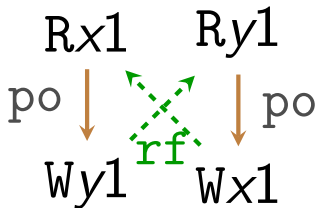
# Load Buffering in C/C++11

$$\begin{array}{l} a := [x]; \\ [y] := a; \end{array} \parallel \begin{array}{l} b := [y]; \\ [x] := b; \end{array}$$

Axioms:

1. **hb** is acyclic

...



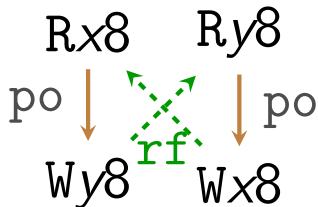
# Load Buffering in C/C++11

$$\begin{array}{l}
 a := [x]; \\
 [y] := a;
 \end{array}
 \parallel
 \parallel
 \begin{array}{l}
 b := [y]; \\
 [x] := b;
 \end{array}$$

Axioms:

1. **hb** is acyclic

...



[OOA-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

	EC	CO	No OOTA
SC MM, [Lamport, 1979]	✗	✗	✓
Java MM, [Manson et al., 2005]	✓	✗	✓
C/C++11 MM, [Batty et al., 2011]	✓	✓*	✗
Proposed solution [Kang et al., 2017] Promise MM, for C/C++ and Java	✓	✓	✓

## Requirements:

- allow **E**fficient **C**ompilation (x86, Power, ARM)
- validate **C**ompiler **O**ptimizations (merging, rearranging, etc)
- no **O**ut-**O**f-**T**hin-**A**ir

# [Store Buffering in Promise]



# Load Buffering in Promise

$$\begin{array}{l} a := [x]; \\ [y] := 1; \end{array} \parallel \begin{array}{l} b := [y]; \\ [x] := 1; \end{array}$$

Memory:  $[\langle x : 0 @ \mathbf{0}_\tau \rangle, \langle y : 0 @ \mathbf{0}_\tau \rangle]$

Final values  $a = \_$ ,  $b = \_$

# Load Buffering in Promise

$$\begin{array}{l} \xrightarrow{\quad} \\ a := [x]; \\ [y] := 1; \end{array} \parallel \begin{array}{l} \xrightarrow{\quad} \\ b := [y]; \\ [x] := 1; \end{array}$$

Memory:  $[\langle x : 0 @ \mathbf{0}_\tau \rangle, \langle y : 0 @ \mathbf{0}_\tau \rangle]$

Final values  $a = \_$ ,  $b = \_$

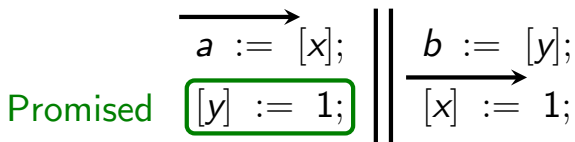
# Load Buffering in Promise

Promised  $\overrightarrow{a := [x];} \parallel \overrightarrow{b := [y];}$   
 $[y] := 1; \parallel [x] := 1;$

Memory:  $\langle x : 0 @ \mathbf{0}_\tau \rangle, \langle y : 0 @ \mathbf{0}_\tau \rangle,$   
 $\langle y : 1 @ \mathbf{1}_\tau \rangle$

Final values  $a = \_$ ,  $b = \_$

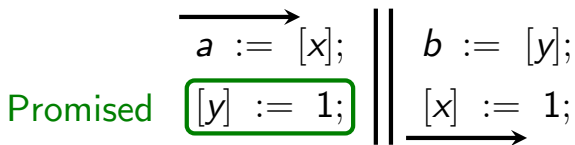
# Load Buffering in Promise



Memory:  $\langle x : 0 @ \mathbf{0}_\tau \rangle, \langle y : 0 @ \mathbf{0}_\tau \rangle,$   
 $\langle y : 1 @ \mathbf{1}_\tau \rangle$

Final values  $a = \_$ ,  $b = \mathbf{1}$

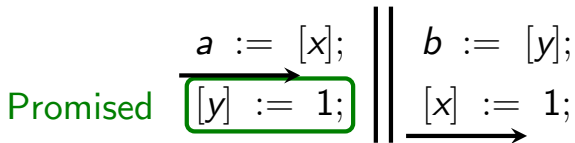
# Load Buffering in Promise



Memory:  $\langle x : 0 @ \mathbf{0}_\tau \rangle, \langle y : 0 @ \mathbf{0}_\tau \rangle,$   
 $\langle y : 1 @ \mathbf{1}_\tau \rangle, \langle x : 1 @ \mathbf{1}_\tau \rangle$

Final values  $a = \_$ ,  $b = \mathbf{1}$

# Load Buffering in Promise



Memory:  $\langle x : 0 @ \mathbf{0}_\tau \rangle, \langle y : 0 @ \mathbf{0}_\tau \rangle,$   
 $\langle y : 1 @ \mathbf{1}_\tau \rangle, \langle x : 1 @ \mathbf{1}_\tau \rangle$

Final values  $a = 1, b = 1$

# Load Buffering in Promise

$$\begin{array}{l}
 a := [x]; \\
 [y] := 1;
 \end{array}
 \parallel
 \begin{array}{l}
 b := [y]; \\
 [x] := 1;
 \end{array}$$

Memory:  $\langle x : 0 @ \mathbf{0}_\tau \rangle, \langle y : 0 @ \mathbf{0}_\tau \rangle,$   
 $\langle y : 1 @ \mathbf{1}_\tau \rangle, \langle x : 1 @ \mathbf{1}_\tau \rangle$

Final values  $a = 1, b = 1$

# ARM-Weak in Promise

$$\begin{array}{l} a := [x]; \\ [x] := 1; \end{array} \parallel \begin{array}{l} b := [x]; \\ [y] := b; \end{array} \parallel \begin{array}{l} c := [y]; \\ [x] := c; \end{array}$$

$$a = 1?$$



# ARM-Weak in Promise

$$\overrightarrow{a := [x];} \quad \left\| \overrightarrow{b := [x];} \quad \left\| \overrightarrow{c := [y];} \right. \\ \left. [x] := 1; \quad [y] := b; \quad [x] := c;$$

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

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

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

# ARM-Weak in Promise

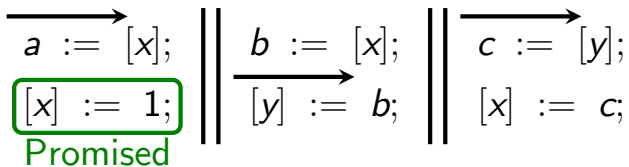
$$\begin{array}{c}
 \xrightarrow{\quad} \\
 a := [x]; \\
 \boxed{[x] := 1;} \\
 \text{Promised}
 \end{array}
 \parallel
 \begin{array}{c}
 \xrightarrow{\quad} \\
 b := [x]; \\
 [y] := b;
 \end{array}
 \parallel
 \begin{array}{c}
 \xrightarrow{\quad} \\
 c := [y]; \\
 [x] := c;
 \end{array}$$

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:       $a = \perp$        $b = \perp$        $c = \perp$

# ARM-Weak in Promise

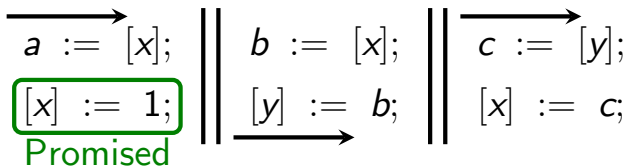


V1:  $[x@0_\tau, y@0_\tau]$     V2:  $[x@2_\tau, y@0_\tau]$     V3:  $[x@0_\tau, y@0_\tau]$

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

Values:             $a = \perp$              $b = 1$              $c = \perp$

# ARM-Weak in Promise

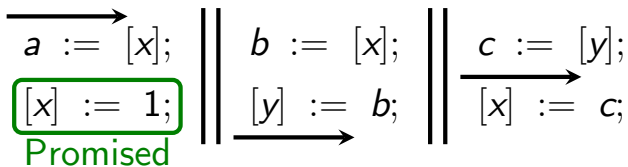


V1:  $[x@0_\tau, y@0_\tau]$     V2:  $[x@2_\tau, y@1_\tau]$     V3:  $[x@0_\tau, y@0_\tau]$

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

Values:             $a = \perp$              $b = 1$              $c = \perp$

# ARM-Weak in Promise

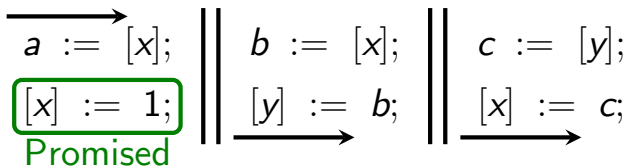


V1:  $[x@0_\tau, y@0_\tau]$     V2:  $[x@2_\tau, y@1_\tau]$     V3:  $[x@0_\tau, y@1_\tau]$

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:             $a = \perp$              $b = 1$              $c = 1$

# ARM-Weak in Promise

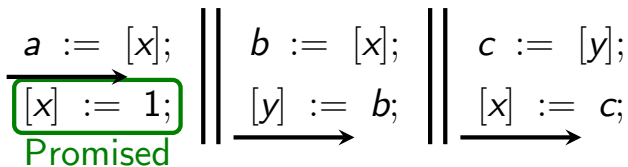


V1:  $[x@0_\tau, y@0_\tau]$     V2:  $[x@2_\tau, y@1_\tau]$     V3:  $[x@1_\tau, y@1_\tau]$

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:             $a = \perp$              $b = 1$              $c = 1$

# ARM-Weak in Promise



V1:  $[x@1_\tau, y@0_\tau]$     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:             $a = 1$              $b = 1$              $c = 1$

# ARM-Weak in Promise

$$\begin{array}{c}
 a := [x]; \\
 [x] := 1;
 \end{array}
 \parallel
 \begin{array}{c}
 b := [x]; \\
 [y] := b;
 \end{array}
 \parallel
 \begin{array}{c}
 c := [y]; \\
 [x] := c;
 \end{array}$$

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

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

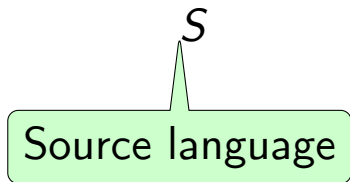
Values:             $a = 1$              $b = 1$              $c = 1$



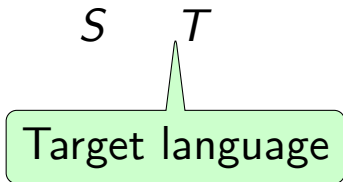
# Agenda

- 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



# Compilation Correctness



# Compilation Correctness

*compile* :  $S \rightarrow T$

# Compilation Correctness

*compile* :  $S \rightarrow T$

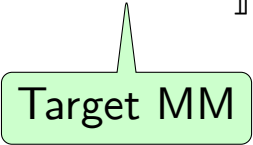
$\llbracket \cdot \rrbracket_S$ .

Source MM

# Compilation Correctness

*compile* :  $S \rightarrow T$

$\llbracket \quad \rrbracket_T$        $\llbracket \quad \rrbracket_S$ .



Target MM

# Compilation Correctness

$compile : S \rightarrow T$

$\forall Prog \in S.$

$\llbracket compile(Prog) \rrbracket_T \subseteq \llbracket Prog \rrbracket_S.$

# 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

- $\text{x86-TSO} = \text{SC} + \text{transformations}$   
[Lahav and Vafeiadis, 2016]

# Compilation to x86-TSO

- $\text{x86-TSO} = \text{SC} + \text{transformations}$   
[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

- $\text{x86-TSO} = \text{SC} + \text{transformations}$   
[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

# Compilation to Power

- Power = StrongPower + transformation  
[Lahav and Vafeiadis, 2016]

# Compilation to Power

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



# Compilation to Power

- Power = StrongPower + transformation  
[Lahav and Vafeiadis, 2016]
  - Reordering of independent instructions
- Transformation is sound in Promise  
[Kang et al., 2017]

# Compilation to Power

- Power = StrongPower + transformation  
[Lahav and Vafeiadis, 2016]
  - Reordering of independent instructions
- Transformation is sound in Promise  
[Kang et al., 2017]
- StrongPower  $\subseteq$  promise-free version of Promise

Scheme isn't applicable to  
ARM POP

# Counterexample. ARM-Weak

$$\begin{array}{l} a := [x]; \text{ // } \mathbf{1} \\ [x] := \mathbf{1}; \end{array} \left\| \left\| \begin{array}{l} b := [x]; \\ [y] := b; \end{array} \right\| \left\| \begin{array}{l} c := [y]; \\ [x] := c; \end{array} \right.$$

# Counterexample. ARM-Weak

$$\begin{array}{l} a := [x]; \text{ // } \mathbf{1} \\ [x] := \mathbf{1}; \end{array} \left\| \left\| \begin{array}{l} b := [x]; \\ [y] := b; \end{array} \right\| \left\| \begin{array}{l} c := [y]; \\ [x] := c; \end{array} \right.$$

**Allowed** in ARM POP

# Counterexample. ARM-Weak

$$\begin{array}{l} a := [x]; \text{ // } \mathbf{1} \\ [x] := \mathbf{1}; \end{array} \left\| \left\| \begin{array}{l} b := [x]; \\ [y] := b; \end{array} \right\| \left\| \begin{array}{l} c := [y]; \\ [x] := c; \end{array} \right.$$

**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

$$\begin{array}{l} a := [x]; \\ b := [y]; \\ c := [x]; \\ [y] := 1; \\ \dots \end{array} \left| \begin{array}{l} \\ \\ \dots \\ \end{array} \right. \begin{array}{l} \\ \\ \dots \\ \end{array}$$

# “Lagging” Simulation

$a := [x];$

$b := [y];$

$c := [x];$

$[y] := 1;$

# “Lagging” Simulation

```
a := [x];
```

```
b := [y];
```

```
c := [x];
```

```
[y] := 1;
```

# “Lagging” Simulation

`a := [x];` ← Fully executed by ARM

`b := [y];`

`c := [x];`

`[y] := 1;`

# “Lagging” Simulation

`a := [x];` ← Fully executed by ARM

`b := [y];`

`c := [x];` ← Partially executed by ARM

`[y] := 1;`

# “Lagging” Simulation

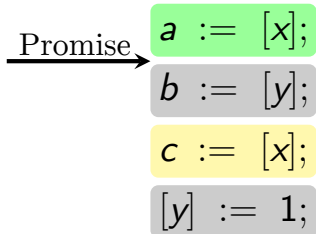
`a := [x];` ← Fully executed by ARM

`b := [y];`

`c := [x];` ← Partially executed by ARM

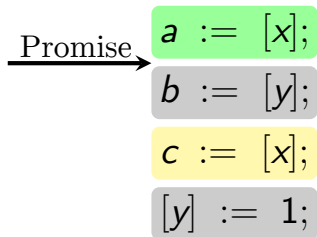
`[y] := 1;` ← Not executed by ARM

# “Lagging” Simulation



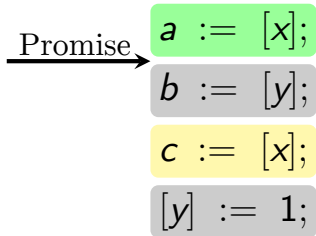


# “Lagging” Simulation



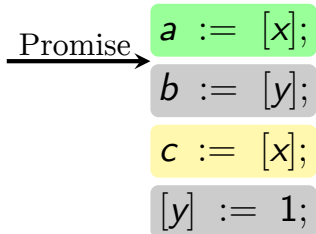
$\mathcal{I}$  — simulation relation

# “Lagging” Simulation



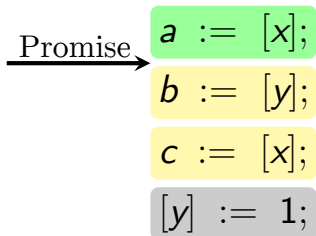
$$\mathcal{I} = \mathcal{I}_{\text{Promise is waiting}} \cup \mathcal{I}_{\text{Promise is executing}}$$

# “Lagging” Simulation



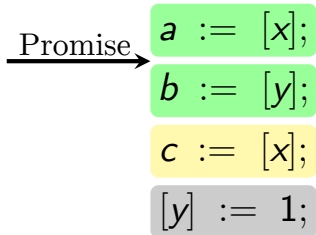
$$\mathcal{I} = \boxed{\mathcal{I}_{\text{Promise is waiting}}} \cup \mathcal{I}_{\text{Promise is executing}}$$

# “Lagging” Simulation



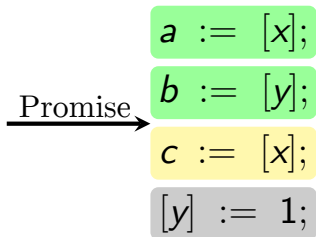
$$\mathcal{I} = \boxed{\mathcal{I}_{\text{Promise is waiting}}} \cup \mathcal{I}_{\text{Promise is executing}}$$

# “Lagging” Simulation



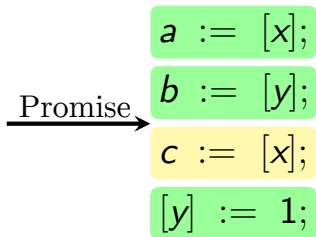
$$\mathcal{I} = \mathcal{I}_{\text{Promise is waiting}} \cup \boxed{\mathcal{I}_{\text{Promise is executing}}}$$

# “Lagging” Simulation



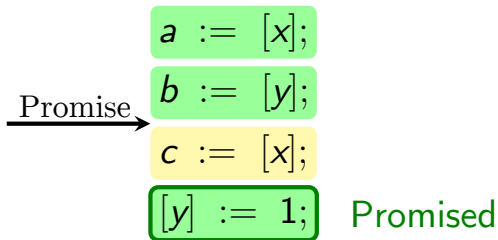
$$\mathcal{I} = \boxed{\mathcal{I}_{\text{Promise is waiting}}} \cup \mathcal{I}_{\text{Promise is executing}}$$

# “Lagging” Simulation



$$\mathcal{I} = \mathcal{I}_{\text{Promise is waiting}} \cup \mathcal{I}_{\text{Promise is executing}}$$

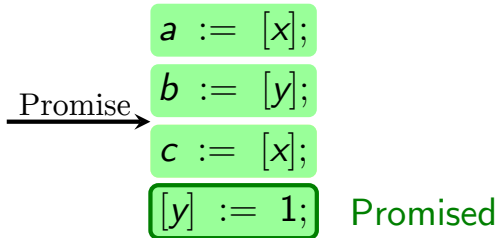
# “Lagging” Simulation



$$\mathcal{I} = \boxed{\mathcal{I}_{\text{Promise is waiting}}} \cup \mathcal{I}_{\text{Promise is executing}}$$

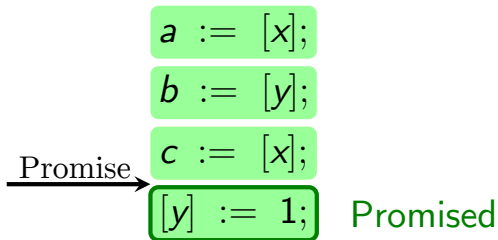


# “Lagging” Simulation



$$\mathcal{I} = \mathcal{I}_{\text{Promise is waiting}} \cup \boxed{\mathcal{I}_{\text{Promise is executing}}}$$

# “Lagging” Simulation



$$\mathcal{I} = \mathcal{I}_{\text{Promise is waiting}} \cup \boxed{\mathcal{I}_{\text{Promise is executing}}}$$

# “Lagging” Simulation

$a := [x];$

$b := [y];$

$c := [x];$

$[y] := 1;$

Promise  
→

$$\mathcal{I} = \boxed{\mathcal{I}_{\text{Promise is waiting}}} \cup \mathcal{I}_{\text{Promise is executing}}$$

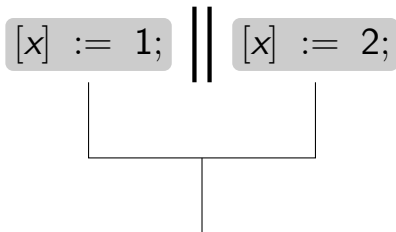
# Main Proof Ingredients

1. “Lagging” simulation
2. ARMv8 POP + timestamps

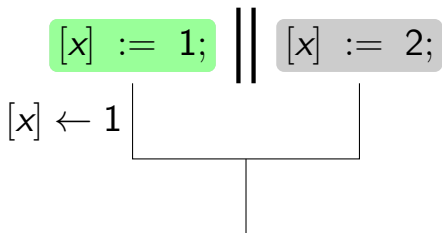
# ARMv8 POP

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

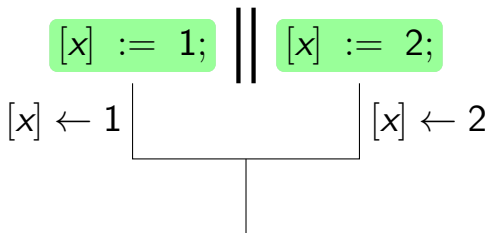
# ARMv8 POP



# ARMv8 POP

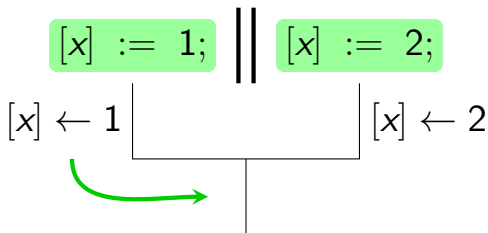


# ARMv8 POP

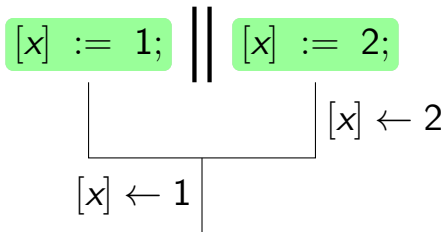




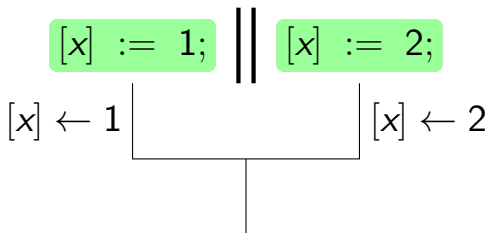
# ARMv8 POP



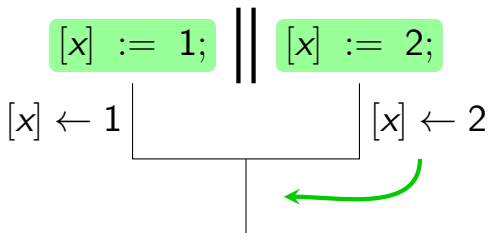
# ARMv8 POP



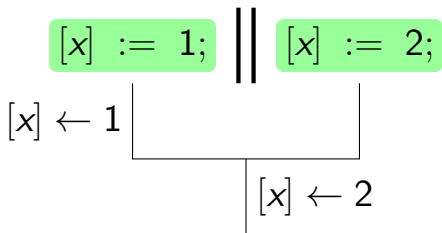
# ARMv8 POP



# ARMv8 POP



# ARMv8 POP

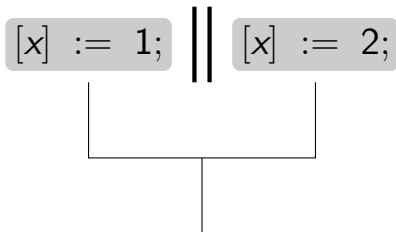


# ARMv8 POP

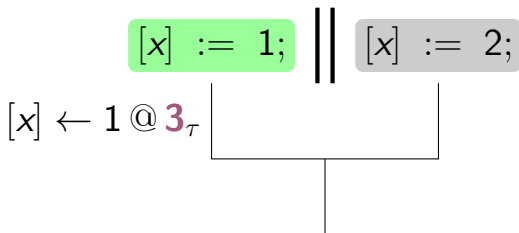
`[x] := 1;` || `[x] := 2;`

Let's determine the order beforehand!

# ARMv8 POP + Timestamps

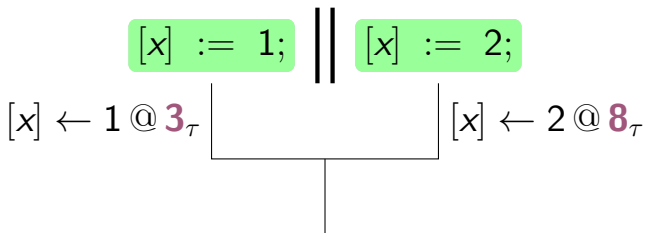


# ARMv8 POP + Timestamps

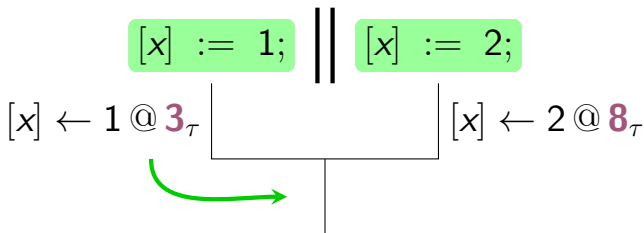




# ARMv8 POP + Timestamps

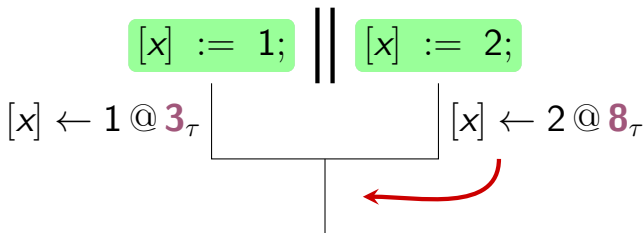


# ARMv8 POP + Timestamps



**May propagate from left**

# ARMv8 POP + Timestamps



**Cannot propagate from right!**

# Proof Structure

1. Introduce  $\text{ARM}+\tau$
2. Prove equivalence between  $\text{ARM}+\tau$  and ARMv8 POP
3. Show “lagging” simulation from Promise to  $\text{ARM}+\tau$

# 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?**

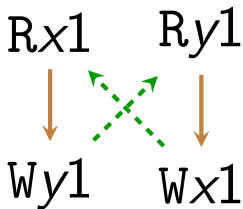


Simulation works for  
operational semantics

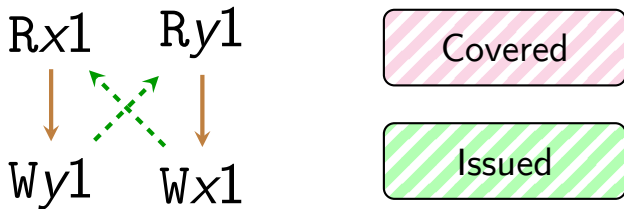
How to **simulate graphs**?

Traverse in proper order!

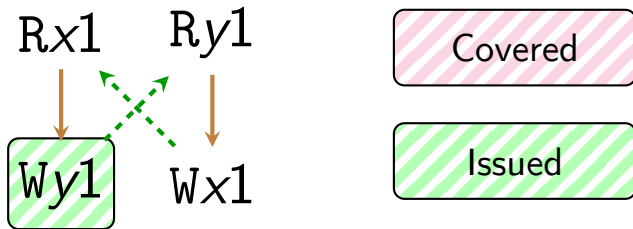
# Traverse of ARMv8.3 execution

$$\begin{array}{l} a := [x]; \\ [y] := 1; \end{array} \parallel \begin{array}{l} b := [y]; \\ [x] := 1; \end{array}$$


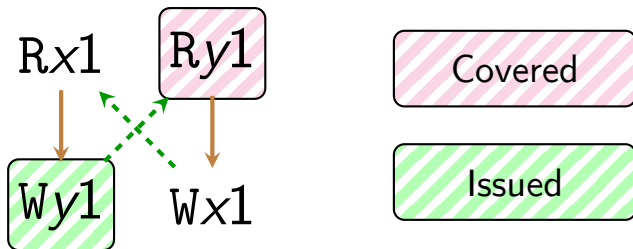
# Traverse of ARMv8.3 execution

$$\begin{array}{l} a := [x]; \\ [y] := 1; \end{array} \parallel \begin{array}{l} b := [y]; \\ [x] := 1; \end{array}$$


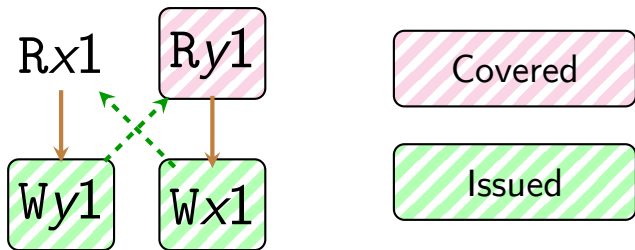
# Traverse of ARMv8.3 execution

$$\begin{array}{l} a := [x]; \\ [y] := 1; \end{array} \parallel \begin{array}{l} b := [y]; \\ [x] := 1; \end{array}$$


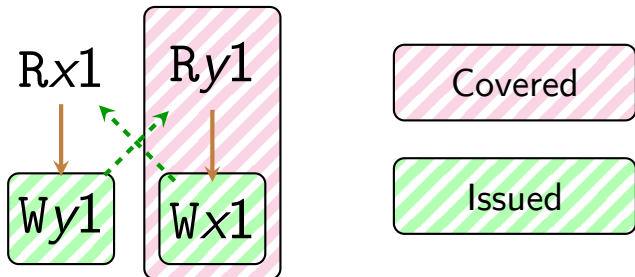
# Traverse of ARMv8.3 execution

$$\begin{array}{l}
 a := [x]; \\
 [y] := 1;
 \end{array}
 \parallel
 \parallel
 \begin{array}{l}
 b := [y]; \\
 [x] := 1;
 \end{array}$$


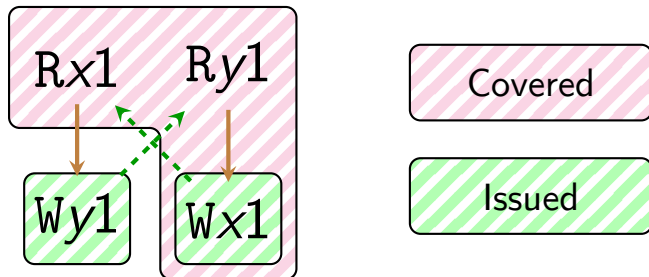
# Traverse of ARMv8.3 execution

$$\begin{array}{l|l} a := [x]; & b := [y]; \\ [y] := 1; & [x] := 1; \end{array}$$


# Traverse of ARMv8.3 execution

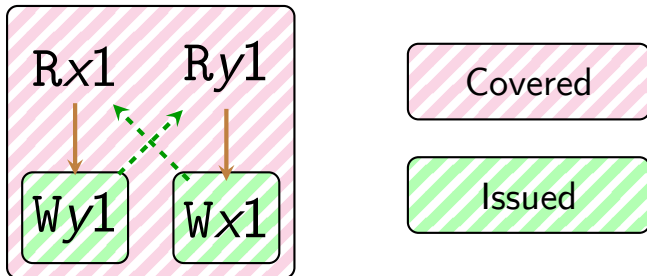
$$\begin{array}{l|l} a := [x]; & b := [y]; \\ [y] := 1; & [x] := 1; \end{array}$$


# Traverse of ARMv8.3 execution

$$\begin{array}{l|l} a := [x]; & b := [y]; \\ [y] := 1; & [x] := 1; \end{array}$$




# Traverse of ARMv8.3 execution

$$\begin{array}{l|l} a := [x]; & b := [y]; \\ [y] := 1; & [x] := 1; \end{array}$$


# Traversal formally

*Cover step:*

$$\frac{\dots}{G \vdash \langle C, I \rangle \rightarrow \langle C \cup \{a\}, I \rangle}$$

*Issue step:*

$$\frac{\dots}{G \vdash \langle C, I \rangle \rightarrow \langle C, I \cup \{w\} \rangle}$$

# Traversal formally

*Cover step:*

$$\frac{\dots}{G \vdash \langle C, I \rangle \rightarrow \langle C \cup \{a\}, I \rangle}$$

Mimics Promise requirements!

*Issue step:*

$$\frac{\dots}{G \vdash \langle C, I \rangle \rightarrow \langle C, I \cup \{w\} \rangle}$$

# 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



Alglave, J., Maranget, L., and Tautschnig, M. (2014).  
Herding cats: Modelling, simulation, testing, and data mining for weak memory.  
*ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74.



Batty, M., Owens, S., Sarkar, S., Sewell, P., and Weber, T. (2011).  
Mathematizing C++ concurrency.  
In *POPL 2011*, pages 55–66. ACM.



Flur, S., Gray, K. E., Pulte, C., Sarkar, S., Sezgin, A., Maranget, L., Deacon, W., and Sewell, P. (2016).  
Modelling the ARMv8 architecture, operationally: Concurrency and ISA.  
In *POPL 2016*, pages 608–621. ACM.



Jeffrey, A. and Riely, J. (2016).  
On thin air reads: Towards an event structures model of relaxed memory.  
In *LICS 2016*. IEEE.



Kang, J., Hur, C.-K., Lahav, O., Vafeiadis, V., and Dreyer, D. (2017).  
A promising semantics for relaxed-memory concurrency.  
In *POPL 2017*. ACM.



Lahav, O. and Vafeiadis, V. (2016).  
Explaining relaxed memory models with program transformations.  
In *FM 2016*. Springer.

## Links II



Lampert, L. (1979).

How to make a multiprocessor computer that correctly executes multiprocess programs.

*IEEE Trans. Computers*, 28(9):690–691.



Manson, J., Pugh, W., and Adve, S. V. (2005).

The Java memory model.

In *POPL 2005*, pages 378–391. ACM.



Owens, S., Sarkar, S., and Sewell, P. (2009).

A better x86 memory model: x86-TSO.

In *TPHOL 2009*, pages 391–407.



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



Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., and Sewell, P. (2018).  
Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models  
for ARMv8.