

# Weak Memory Concurrency in C/C++11 and LLVM

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

March 2017

# Quiz #1. Should these transformations be allowed?

## 1. CSE over acquiring a lock:

$a = x;$		$a = x;$
$lock();$	$\rightsquigarrow$	$lock();$
$b = x;$		$b = a;$

## 2. Load hoisting:

$if(c)$		$t = x;$
$a = x;$	$\rightsquigarrow$	$a = c ? t : a;$

[ $x$  is a global variable;  $a, b, c$  are local;  $t$  is a fresh temporary.]

## Allowing both is clearly wrong!

Consider the transformation sequence:

<code>if (c)</code>		<code>t = x;</code>		<code>t = x;</code>
<code>  a = x;</code>	<i>hoist</i>	<code>a = c ? t : a;</code>	<i>CSE</i>	<code>a = c ? t : a;</code>
<code>  lock();</code>		<code>  lock();</code>		<code>  lock();</code>
<code>  b = x;</code>		<code>  b = x;</code>		<code>  b = t;</code>

When  $c$  is false,  $x$  is moved out of the critical region!

So we have to forbid one transformation.

- ▶ C11 forbids load hoisting, allows CSE over `lock()`.
- ▶ LLVM allows load hoisting, forbids CSE over `lock()`.

## **Unambiguous specification**

- ▶ Which are the possible outcomes of a program.
- ▶ Which optimizations may the compiler perform.

## **Typically called a **weak memory model (WMM)****

- ▶ Allows more behaviors than thread interleaving.

## **Amenable to formal reasoning**

- ▶ Can prove theorems about the model.
- ▶ Objectively compare memory models.

## Unambiguous specification

- ▶ Which are the possible outcomes of a program.
- ▶ Which optimizations may the compiler perform.

## Typically called a **weak memory model (WMM)**

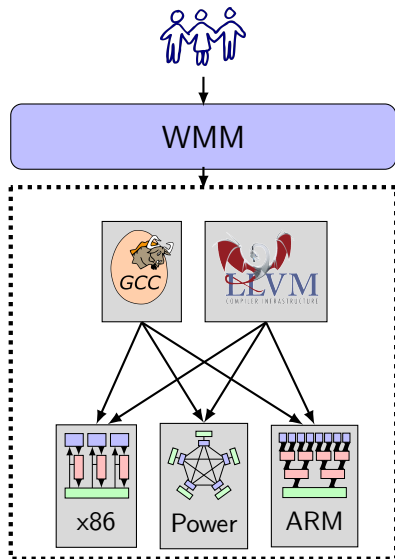
- ▶ Allows more behaviors than thread interleaving.

## Amenable to formal reasoning

- ▶ Can prove theorems about the model.
- ▶ Objectively compare memory models.

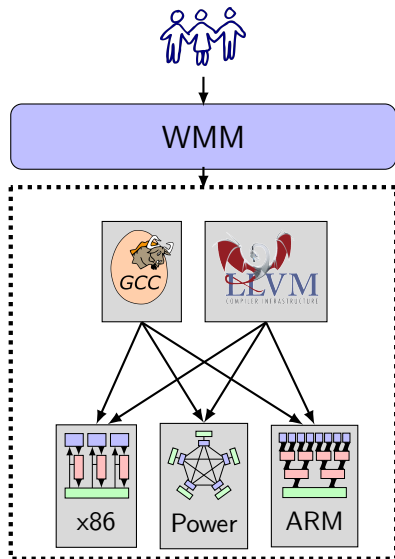
## **But it is not easy to get right**

- ▶ The Java memory model is flawed.
- ▶ The C/C++11 model is also flawed.



## WMM desiderata

1. Mathematically sane  
(e.g., monotone)
2. Not too weak  
(good for programmers)
3. Not too strong  
(good for hardware)
4. Admits optimizations  
(good for compilers :-)



## Outline

- ▶ How to define a weak memory model?
- ▶ The C/C++ memory model (a.k.a. C11)
- ▶ Unfortunate flaws in C11
- ▶ The OOTA problem
- ▶ A 'promising' solution

# Three approaches for defining WMMs

## **Operational**

- ▶ Define program semantics with an abstract machine.

## **Transformational**

- ▶ Define the model as a sequence of program transformations over some basic model (e.g., SC).

## **Axiomatic**

- ▶ Define the model as a set of consistency constraints on program *executions*.

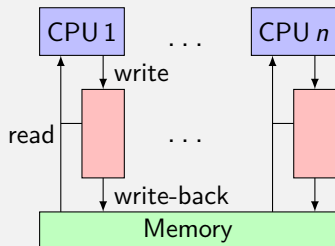


# Operational approach

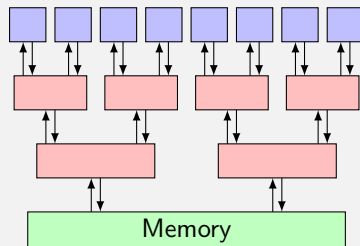
Define program semantics with an abstract machine.

- ▶ Works well for most hardware models.
- ▶ Very low-level  $\leadsto$  cumbersome to reason about.
- ▶ May require elaborate features for PL models.

## x86-TSO model (2010)



## ARMv8 model (2016)



# Transformational approach

Define the model as a sequence of program transformations over some basic operational model, such as SC.

For example,

$$\text{TSO} = \text{SC} + \text{WR-reordering} + \text{RaW-elimination}$$

# Transformational approach

Define the model as a sequence of program transformations over some basic operational model, such as SC.

For example,

$$\text{TSO} = \text{SC} + \text{WR-reordering} + \text{RaW-elimination}$$

**But:**

- ▶ Applicable only in very few cases.
- ▶ Does not work for ARM.

ARM weak

$$\begin{array}{l} a = x; \text{ // } 1 \\ x = 1; \end{array} \parallel y = x; \parallel x = y;$$

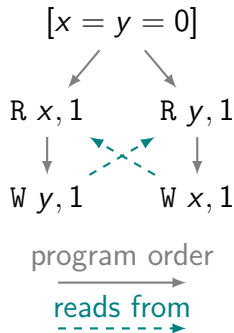
# Axiomatic approach

Define the model as a set of consistency constraints on program executions.

Example: Load-buffering

```
a = x; // 1   ||   b = y; // 1
y = 1;       ||   x = b;
```

- ▶ Works well for hardware models.
- ▶ Followed by C11.
- ▶ Problematic for programming languages because of **OOTA** (“out of thin air”) values.



## The C11 memory model

- ▶ Introduced by the ISO C/C++ 2011 standards.
- ▶ Defines the semantics of **concurrent** memory accesses.
- ▶ Adopted by the LLVM IR with some changes.  
(The differences are not relevant for this talk.)

# The C11 memory model: Atomics

Two types of locations

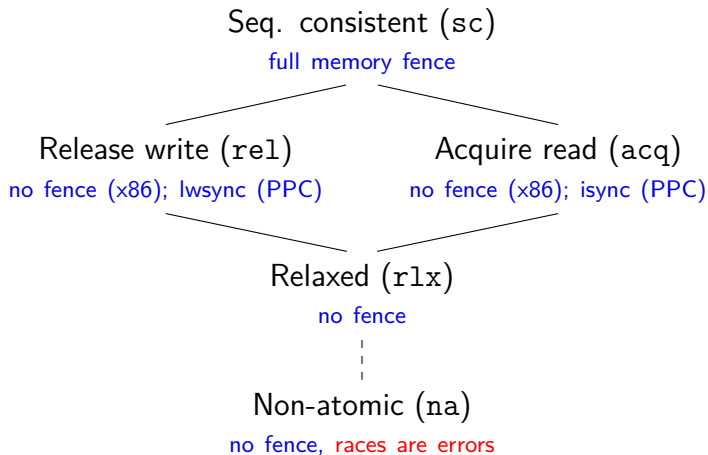
**Ordinary  
(Non-Atomic)**

Races are **errors**

**Atomic**

Welcome to the  
**expert mode**

# A spectrum of accesses



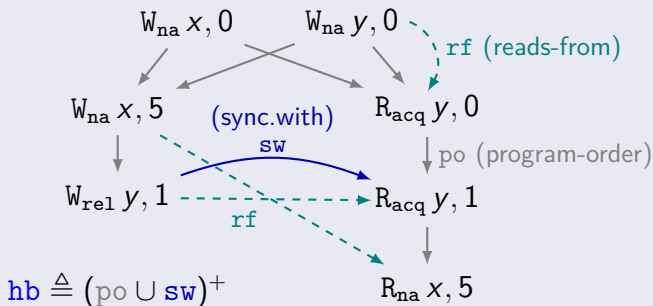
Explicit primitives for fences

# An execution in C11: actions and relations (and axioms)

Initially  $x = y = 0$ .

```
x = 5;
y.store(1, release);  |||  while (y.load(acq) == 0);
                        |||  print(x);
```

## One possible execution





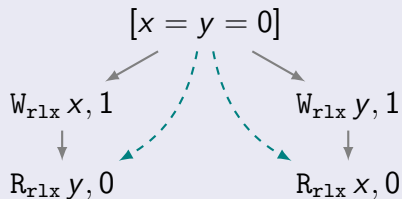
## Relaxed behavior: store buffering

Initially  $x = y = 0$ .

```
x.store(1, rlx);    || y.store(1, rlx);  
a = y.load(rlx);   || b = x.load(rlx);
```

This can return  $a = b = 0$ .

### Justification



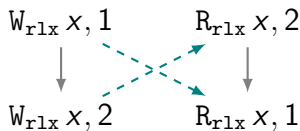
Behavior observed on  
x86/Power/ARM

# Coherence

Programs with a single shared variable behave as under SC.

```
x.store(1, rlx);  ||  a = x.load(rlx);  
x.store(2, rlx);  ||  b = x.load(rlx);
```

The outcome  $a = 2 \wedge b = 1$  is forbidden.

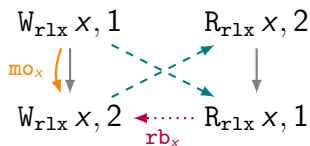


# Coherence

Programs with a single shared variable behave as under SC.

$$\begin{array}{l} x.\text{store}(1, rlx); \\ x.\text{store}(2, rlx); \end{array} \parallel \begin{array}{l} a = x.\text{load}(rlx); \\ b = x.\text{load}(rlx); \end{array}$$

The outcome  $a = 2 \wedge b = 1$  is forbidden.



- ▶ Modification order,  $mo_x$ , total order of writes to  $x$ .
- ▶ Reads-before :  $rb_x \triangleq (rf_x^{-1}; mo_x) \cap (\neq)$
- ▶ Coherence :  $hb \cup rf_x \cup mo_x \cup rb_x$  is acyclic for all  $x$ .

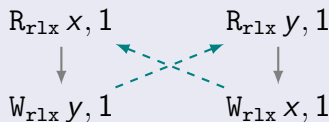
## Causality cycles with relaxed accesses

Initially  $x = y = 0$ .

```
if (x.load(rlx) == 1) || if (y.load(rlx) == 1)
    y.store(1, rlx);      x.store(1, rlx);
```

C11 allows the outcome  $x = y = 1$ .

### Justification



Relaxed accesses don't  
synchronize

## No causality cycles with non-atomics

Initially  $x = y = 0$ .

$$\begin{array}{l} \mathbf{if} (x == 1) \\ \quad y = 1; \end{array} \parallel \begin{array}{l} \mathbf{if} (y == 1) \\ \quad x = 1; \end{array}$$

C11 forbids the outcome  $x = y = 1$ .

### Justification

Non-atomic read axiom:

$$\mathbf{rf} \cap (\_ \times \mathbf{NA}) \subseteq \mathbf{hb}$$

## Is the C11 memory model definition...

1. Mathematically sane?
  - ▶ *For example, it is monotone.*
2. Not too weak?
  - ▶ *Provides useful reasoning principles.*
3. Not too strong?
  - ▶ *Can be implemented efficiently.*
4. Actually useful?
  - ▶ *Admits the intended program optimizations.*

## Is the C11 memory model definition...

1. Mathematically sane?
  - ▶ *For example, it is monotone.*
2. Not too weak?
  - ▶ *Provides useful reasoning principles.*
3. Not too strong?
  - ✓ *Compilation to x86/Power/ARM.*
4. Actually useful?
  - ▶ *Admits the intended program optimizations.*

# Is the C11 memory model definition...

1. Mathematically sane?

- ▶ *For example, it is monotone.*

2. Not too weak?

≈ Reasoning principles for C11 subsets.

3. Not too strong?

✓ Compilation to x86/Power/ARM.

4. Actually useful?

- ▶ *Admits the intended program optimizations.*



## Is the C11 memory model definition...

1. Mathematically sane?

✗ No, it is not monotone.

2. Not too weak?

≈ Reasoning principles for C11 subsets.

3. Not too strong?

✓ Compilation to x86/Power/ARM.

4. Actually useful?

▶ *Admits the intended program optimizations.*

## Is the C11 memory model definition...

1. Mathematically sane?

✗ No, it is not monotone.

2. Not too weak?

≈ Reasoning principles for C11 subsets.

3. Not too strong?

✓ Compilation to x86/Power/ARM.

4. Actually useful?

✗ No, it disallows intended program transformations.

## Is the C11 memory model definition...

1. Mathematically sane?

✗ No, it is not monotone.

2. Not too weak?

≈ Reasoning principles for C11 subsets.

3. Not too strong?

✗ Compilation to Power and ARM is broken.

4. Actually useful?

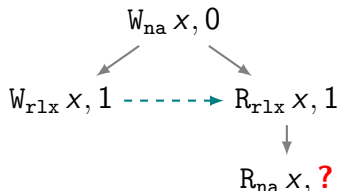
✗ No, it disallows intended program transformations.

# Non-atomic reads of atomic variables are unsound!

Initially,  $x = 0$ .

```
x.store(1, r/x); || if (x.load(r/x) == 1)
                   t = (int) x;
```

The program can get stuck!



- ▶ Reading 0 contradicts coherence.
- ▶ Reading 1 contradicts the **non-atomic read axiom**.

## Sequentialization is invalid

Initially,  $a = x = y = 0$ .

$$a = 1; \left\| \begin{array}{l} \text{if } (x.\text{load}(r/x) == 1) \\ \quad \text{if } (a == 1) \\ \quad \quad y.\text{store}(1, r/x); \end{array} \right\| \left\| \begin{array}{l} \text{if } (y.\text{load}(r/x) == 1) \\ \quad x.\text{store}(1, r/x); \end{array} \right\|$$

The only possible output is:

$$a = 1, \quad x = y = 0.$$

Recall the non-atomic read axiom:

$$\text{rf} \cap (\_ \times \text{NA}) \subseteq \text{hb}$$

## Tentative fixes

Remove non-atomic read axiom.

- ▶ gives extremely weak guarantees, if any

In addition, forbid  $(po \cup rf)$ -cycles.

- ▶ rules out causal loops
- ▶ forbids some reorderings
- ▶ more costly on ARM/Power

Related to the OOTA problem...

- ▶ More in a couple of slides

“Adding synchronization should not introduce new behaviors”

Examples:

- ▶ Reducing parallelism,  $C_1 \parallel C_2 \rightsquigarrow C_1 ; C_2$
- ▶ Expression evaluation linearization:

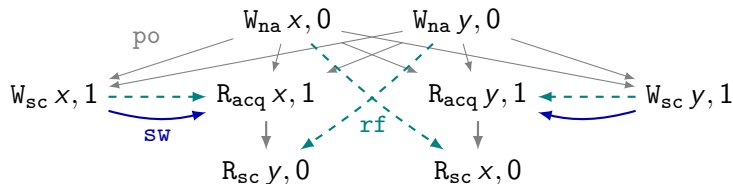
$$x = a + b ; \rightsquigarrow t_1 = a ; t_2 = b ; x = t_1 + t_2 ;$$

- ▶ Adding a memory fence
- ▶ Strengthening the access mode of an operation
- ▶ (Roach motel reorderings)

## IRIW-acq-sc

$$x_{sc} = 1; \parallel \begin{array}{l} a = x_{acq}; // 1 \\ c = y_{sc}; // 0 \end{array} \parallel \begin{array}{l} b = y_{acq}; // 1 \\ d = x_{sc}; // 0 \end{array} \parallel y_{sc} = 1;$$

- ▶ C11 **disallows** the annotated behavior:



- ▶ The behavior is, however, **allowed** on POWER/ARM following the “trailing sync” compilation scheme.



The axiom of SC reads is too weak.

- ▶ Makes strengthening unsound.

The axioms of SC fences are too weak.

- ▶ They do not guarantee sequential consistency.

The definition of release sequences is too strong.

- ▶ Removing  $(po \cup rf)$ -final events is unsound.

## The OOTA problem

# The *out-of-thin-air* problem in C11

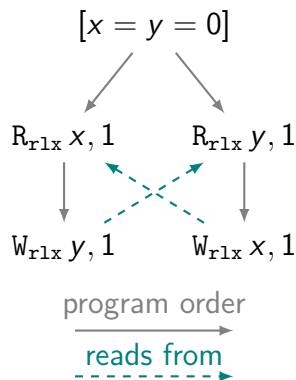
- ▶ Initially,  $x = y = 0$ .
- ▶ All accesses are “relaxed”.

## Load-buffering

```
a = x; // 1    ||    x = y;  
y = 1;
```

This behavior must be allowed:

Power/ARM allow it



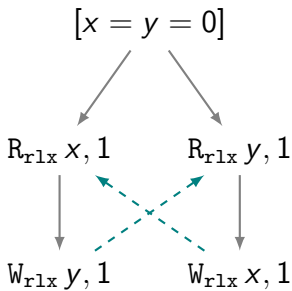
# The *out-of-thin-air* problem in C11

Load-buffering + data dependency

```
a = x; // 1 || x = y;  
y = a;
```

The behavior should be forbidden:

**Values appear out-of-thin-air!**



Same execution as before!  
C11 allows these behaviors

# The *out-of-thin-air* problem in C11

## Load-buffering + data dependency

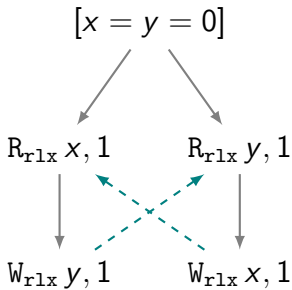
```
a = x; // 1 || x = y;  
y = a;
```

The behavior should be forbidden:  
**Values appear out-of-thin-air!**

## Load-buffering + control dependencies

```
a = x; // 1 || if (y == 1)  
if (a == 1)    x = 1;  
y = 1;
```

The behavior should be forbidden:  
**DRF guarantee is broken!**



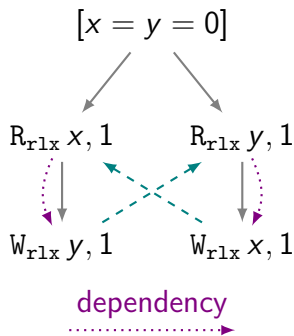
Same execution as before!  
C11 allows these behaviors

# The hardware solution

Keep track of syntactic dependencies, and forbid “dependency cycles”.

Load-buffering + data dependency

```
a = x; // 1
y = a;
||
x = y;
```



# The hardware solution

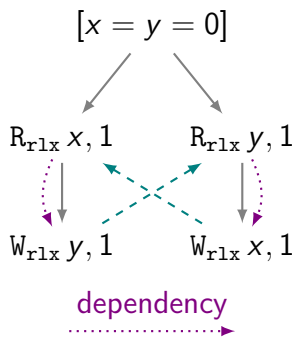
Keep track of syntactic dependencies, and forbid “dependency cycles”.

Load-buffering + data dependency

```
a = x; // 1      ||      x = y;  
y = a;
```

Load-buffering + fake dependency

```
a = x; // 1      ||      x = y;  
y = a + 1 - a;
```



This approach is not suitable for a programming language:  
**Compilers do not preserve syntactic dependencies.**

## A ‘promising’ solution to OOTA

We propose a model that satisfies all WMM desiderata, and covers nearly all features of C11.

- ▶ No “out-of-thin-air” values
- ▶ DRF guarantees
- ▶ Efficient h/w mappings
- ▶ Compiler optimizations

**Key idea:** Start with an operational interleaving semantics, but allow threads to **promise** to write in the future.



# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
      x = y = 0
x = 1;  ||  y = 1;
a = y; // 0 || b = x; // 0
```

# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
x = y = 0
┆
▶ x = 1;   │   │   ▶ y = 1;
a = y; // 0 │   │   b = x; // 0
```

## Memory

```
⟨x : 0@0⟩
⟨y : 0@0⟩
```

## $T_1$ 's view

x	y
0	0

## $T_2$ 's view

x	y
0	0

- ▶ Global memory is a pool of messages of the form

$\langle location : value @ timestamp \rangle$

- ▶ Each thread maintains a *thread-local view* recording the last observed timestamp for every location

# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
x = y = 0
x = 1;      |      y = 1;
▶ a = y; // 0 |      b = x; // 0
```

## Memory

```
⟨x : 0@0⟩
⟨y : 0@0⟩
⟨x : 1@1⟩
```

## $T_1$ 's view

x	y
<del>0</del>	0
1	

## $T_2$ 's view

x	y
0	0

- ▶ Global memory is a pool of messages of the form

$\langle location : value @ timestamp \rangle$

- ▶ Each thread maintains a *thread-local view* recording the last observed timestamp for every location

# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
      x = y = 0
x = 1;  ||  y = 1;
▶ a = y; // 0  ||  ▶ b = x; // 0
```

## Memory

```
⟨x : 0@0⟩
⟨y : 0@0⟩
⟨x : 1@1⟩
⟨y : 1@1⟩
```

## $T_1$ 's view

x	y
<del>0</del>	0
1	

## $T_2$ 's view

x	y
0	<del>1</del>
	1

- ▶ Global memory is a pool of messages of the form

$\langle location : value @ timestamp \rangle$

- ▶ Each thread maintains a *thread-local view* recording the last observed timestamp for every location

# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
x = y = 0
x = 1;          y = 1;
a = y; // 0    ▶ b = x; // 0
```

## Memory

```
⟨x : 0@0⟩
⟨y : 0@0⟩
⟨x : 1@1⟩
⟨y : 1@1⟩
```

## $T_1$ 's view

x	y
<del>x</del>	0
1	

## $T_2$ 's view

x	y
0	<del>y</del>
	1

- ▶ Global memory is a pool of messages of the form

$\langle location : value @ timestamp \rangle$

- ▶ Each thread maintains a *thread-local view* recording the last observed timestamp for every location

# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
      x = y = 0
x = 1;  ||  y = 1;
a = y; // 0 || b = x; // 0
```

## Memory

```
⟨x : 0@0⟩
⟨y : 0@0⟩
⟨x : 1@1⟩
⟨y : 1@1⟩
```

## $T_1$ 's view

x	y
<del>0</del>	0
1	

## $T_2$ 's view

x	y
0	<del>1</del>
	1

- ▶ Global memory is a pool of messages of the form

$\langle location : value @ timestamp \rangle$

- ▶ Each thread maintains a *thread-local view* recording the last observed timestamp for every location

# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
      x = y = 0
x = 1;  ||  y = 1;
a = y; // 0 || b = x; // 0
```

## Memory

```
⟨x : 0@0⟩
⟨y : 0@0⟩
⟨x : 1@1⟩
⟨y : 1@1⟩
```

## $T_1$ 's view

x	y
<del>0</del>	0
1	

## $T_2$ 's view

x	y
0	<del>1</del>
	1

## Coherence Test

```
      x = 0
x = 1;  ||  x = 2;
a = x; // 2 || b = x; // 1
```

# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
      x = y = 0
x = 1;  ||  y = 1;
a = y; // 0 || b = x; // 0
```

## Memory

```
⟨x : 0@0⟩
⟨y : 0@0⟩
⟨x : 1@1⟩
⟨y : 1@1⟩
```

## $T_1$ 's view

x	y
<del>x</del>	0
1	

## $T_2$ 's view

x	y
0	<del>y</del>
	1

## Coherence Test

```
      x = 0
x = 1;  ||  x = 2;
a = x; // 2 || b = x; // 1
```

## Memory

```
⟨x : 0@0⟩
```

## $T_1$ 's view

x
0

## $T_2$ 's view

x
0



# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
      x = y = 0
x = 1;  ||  y = 1;
a = y; // 0 || b = x; // 0
```

## Memory

```
⟨x : 0@0⟩
⟨y : 0@0⟩
⟨x : 1@1⟩
⟨y : 1@1⟩
```

## $T_1$ 's view

x	y
<del>0</del>	0
1	

## $T_2$ 's view

x	y
0	<del>1</del>
	1

## Coherence Test

```
      x = 0
x = 1;  ||  x = 2;
a = x; // 2 || b = x; // 1
```

## Memory

```
⟨x : 0@0⟩
⟨x : 1@1⟩
```

## $T_1$ 's view

x
<del>0</del>
1

## $T_2$ 's view

x
0

# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
x = y = 0
x = 1;          ||      y = 1;
a = y; // 0    ||      b = x; // 0
```

## Memory

```
<x : 0@0>
<y : 0@0>
<x : 1@1>
<y : 1@1>
```

## $T_1$ 's view

x	y
<del>0</del>	0
1	

## $T_2$ 's view

x	y
0	<del>0</del>
	1

## Coherence Test

```
x = 0
x = 1;          ||      x = 2;
a = x; // 2    ||      b = x; // 1
```

## Memory

```
<x : 0@0>
<x : 1@1>
<x : 2@2>
```

## $T_1$ 's view

x
<del>0</del>
1

## $T_2$ 's view

x
<del>0</del>
2

# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
      x = y = 0
x = 1;  ||  y = 1;
a = y; // 0 || b = x; // 0
```

## Memory

```
<x : 0@0>
<y : 0@0>
<x : 1@1>
<y : 1@1>
```

## $T_1$ 's view

x	y
<del>x</del>	0
1	

## $T_2$ 's view

x	y
0	<del>y</del>
	1

## Coherence Test

```
      x = 0
x = 1;  ||  x = 2;
a = x; // 2  ▶ b = x; // 1
```

## Memory

```
<x : 0@0>
<x : 1@1>
<x : 2@2>
```

## $T_1$ 's view

x
<del>x</del>
<del>x</del>
2

## $T_2$ 's view

x
<del>x</del>
2

# Simple operational semantics for C11's relaxed accesses

## Store-buffering

```
      x = y = 0
x = 1;  ||  y = 1;
a = y; // 0 || b = x; // 0
```

## Memory

```
<x : 0@0>
<y : 0@0>
<x : 1@1>
<y : 1@1>
```

## $T_1$ 's view

x	y
<del>x</del>	0
1	

## $T_2$ 's view

x	y
0	<del>y</del>
	1

## Coherence Test

```
      x = 0
x = 1;  ||  x = 2;
a = x; // 2 || b = x; // 1
```

## Memory

```
<x : 0@0>
<x : 1@1>
<x : 2@2>
```

## $T_1$ 's view

x
<del>x</del>
<del>x</del>
2

## $T_2$ 's view

x
<del>x</del>
2

## Load-buffering

```
x = y = 0  
a = x; // 1  
y = 1;  || x = y;
```

- ▶ To model load-store reordering, we allow **“promises”**.
- ▶ At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.

# Promises

## Load-buffering

```
x = y = 0
▶ a = x; // 1 || ▶ x = y;
y = 1;
```

## Memory

```
⟨x : 0@0⟩
⟨y : 0@0⟩
```

## $T_1$ 's view

x	y
0	0

## $T_2$ 's view

x	y
0	0

- ▶ To model load-store reordering, we allow **“promises”**.
- ▶ At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.

# Promises

## Load-buffering

```
x = y = 0
▶ a = x; // 1 || ▶ x = y;
y = 1;
```

## Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@1 \rangle$

## $T_1$ 's view

x	y
0	0

## $T_2$ 's view

x	y
0	0

- ▶ To model load-store reordering, we allow **“promises”**.
- ▶ At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.

# Promises

## Load-buffering

```
x = y = 0
▶ a = x; // 1  ||  ▶ x = y;
y = 1;
```

## Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@1 \rangle$

## $T_1$ 's view

x	y
0	0

## $T_2$ 's view

x	y
0	<del>0</del>
	1

- ▶ To model load-store reordering, we allow **“promises”**.
- ▶ At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.



# Promises

## Load-buffering

```
x = y = 0  
▶ a = x; // 1 || x = y;  
y = 1;      ▶
```

## Memory

```
<x : 0@0>  
<y : 0@0>  
<y : 1@1>  
<x : 1@1>
```

## T<sub>1</sub>'s view

x	y
0	0

## T<sub>2</sub>'s view

x	y
<del>0</del>	<del>0</del>
1	1

- ▶ To model load-store reordering, we allow **“promises”**.
- ▶ At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.

# Promises

## Load-buffering

```
x = y = 0
a = x; // 1
y = 1;
x = y;
```

## Memory

```
<x : 0@0>
<y : 0@0>
<y : 1@1>
<x : 1@1>
```

## T<sub>1</sub>'s view

x	y
<del>0</del>	0
1	

## T<sub>2</sub>'s view

x	y
<del>0</del>	<del>0</del>
1	1

- ▶ To model load-store reordering, we allow **“promises”**.
- ▶ At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.

# Promises

## Load-buffering

```
x = y = 0
a = x; // 1
y = 1;
      ||
      x = y;
```

## Memory

```
<x : 0@0>
<y : 0@0>
<y : 1@1>
<x : 1@1>
```

## T<sub>1</sub>'s view

x	y
<del>0</del>	<del>0</del>
1	1

## T<sub>2</sub>'s view

x	y
<del>0</del>	<del>0</del>
1	1

- ▶ To model load-store reordering, we allow **“promises”**.
- ▶ At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.

# Promises

## Load-buffering

```
      x = y = 0
a = x; // 1  |||  x = y;
y = 1;      |||
▶           ▶
```

## Load-buffering + dependency

```
  a = x; // 1  |||  x = y;
  y = a;      |||
```

## Memory

```
⟨x : 0@0⟩
⟨y : 0@0⟩
⟨y : 1@1⟩
⟨x : 1@1⟩
```

## $T_1$ 's view

x	y
<del>0</del>	<del>0</del>
1	1

## $T_2$ 's view

x	y
<del>0</del>	<del>0</del>
1	1

Must not admit the same execution!

# Promises

## Load-buffering

```
x = y = 0  
a = x; // 1  
y = 1;  || x = y;  
▶      ▶
```

## Load-buffering + dependency

```
a = x; // 1  
y = a;  || x = y;
```

## Key Idea

A thread can only promise if it can perform the write anyway (even without having made the promise)

### Thread-local certification

A thread can promise to write a message, if it can *thread-locally certify* that its promise will be fulfilled.

# Certified promises

## Thread-local certification

A thread can promise to write a message, if it can *thread-locally certify* that its promise will be fulfilled.

### Load-buffering

$$\begin{array}{l} a = x; \text{ // } 1 \\ y = 1; \end{array} \parallel x = y;$$

$T_1$  **may promise**  $y = 1$ , since it is able to write  $y = 1$  by itself.

### Load buffering + fake dependency

$$\begin{array}{l} a = x; \text{ // } 1 \\ y = a + 1 - a; \end{array} \parallel x = y;$$

$T_1$  **may NOT promise**  $y = 1$ , since it is not able to write  $y = 1$  by itself.

### Load buffering + dependency

$$\begin{array}{l} a = x; \text{ // } 1 \\ y = a; \end{array} \parallel x = y;$$

Is this behavior possible?

```
a = x; // 1  
x = 1;
```



Is this behavior possible?

```
a = x; // 1  
x = 1;
```

**No.**

Suppose the thread promises  $x = 1$ . Then, once  $a = x$  reads 1, the thread view is increased and so the promise cannot be fulfilled.

## Quiz #3

Is this behavior possible?

```
a = x; // 1 || y = x; || x = y;  
x = 1;
```

Is this behavior possible?

```
a = x; // 1 || y = x; || x = y;  
x = 1;
```

**Yes. And the ARM model allows it!**

Is this behavior possible?

$$\begin{array}{l} a = x; \text{ // } 1 \\ x = 1; \end{array} \parallel y = x; \parallel x = y;$$

**Yes. And the ARM model allows it!**

This behavior can be also explained by sequentialization:

$$\begin{array}{l} a = x; \text{ // } 1 \\ x = 1; \end{array} \parallel y = x; \parallel x = y; \quad \rightsquigarrow \quad \begin{array}{l} a = x; \text{ // } 1 \\ x = 1; \\ y = x; \end{array} \parallel x = y;$$

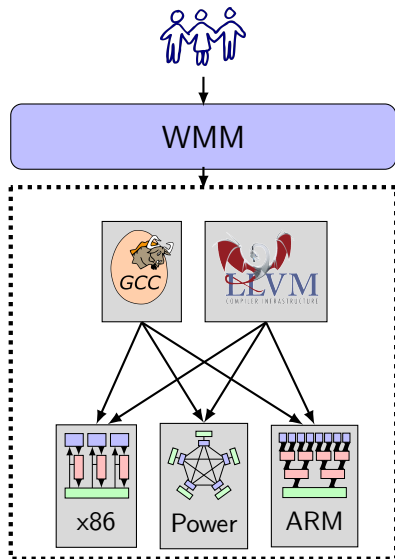
We have extended this basic idea to handle:

- ▶ Atomic updates (e.g., CAS, fetch-and-add)
- ▶ Release/acquire fences and accesses
- ▶ Release sequences
- ▶ SC fences
- ▶ Plain accesses (C11's non-atomics & Java's normal accesses)

## Results

- ▶ No “out-of-thin-air” values
- ▶ DRF guarantees
- ▶ Efficient h/w mappings (x86-TSO, Power, ARM)
- ▶ Compiler optimizations (incl. reorderings, eliminations)

# Summary



## Summary

- ▶ The need for a WMM.
- ▶ C11 is **very broken**.
- ▶ Many of the problems are **locally fixable**.
- ▶ But ruling out **OOA** requires an entirely different approach.
- ▶ The **promising model** may be the solution.