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:**

$$\begin{array}{l} a = x; \\ lock(); \\ b = x; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} a = x; \\ lock(); \\ b = a; \end{array}$$

2. **Load hoisting:**

$$\begin{array}{l} \text{if } (c) \\ \quad a = x; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t = x; \\ a = c \ ? \ t : a; \end{array}$$

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

Allowing both is clearly wrong!

Consider the transformation sequence:

$$\begin{array}{l} \text{if } (c) \\ \quad a = x; \\ lock(); \\ b = x; \end{array} \quad \overset{\text{hoist}}{\rightsquigarrow} \quad \begin{array}{l} t = x; \\ a = c \ ? \ t : a; \\ lock(); \\ b = x; \end{array} \quad \overset{\text{CSE}}{\rightsquigarrow} \quad \begin{array}{l} t = x; \\ a = c \ ? \ t : a; \\ lock(); \\ b = t; \end{array}$$

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

So we have to forbid one transfomation.

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

Formal model

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.

Formal model

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.

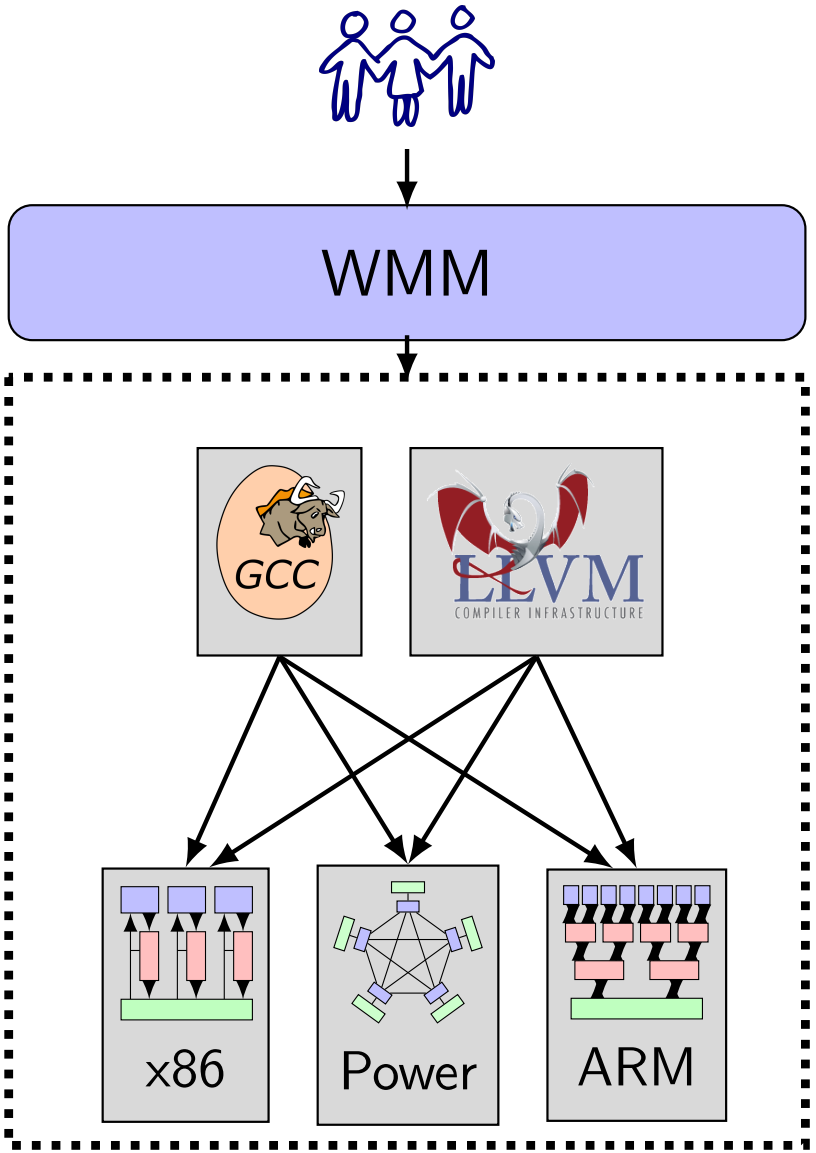
Amenable to formal reasoning

- 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.

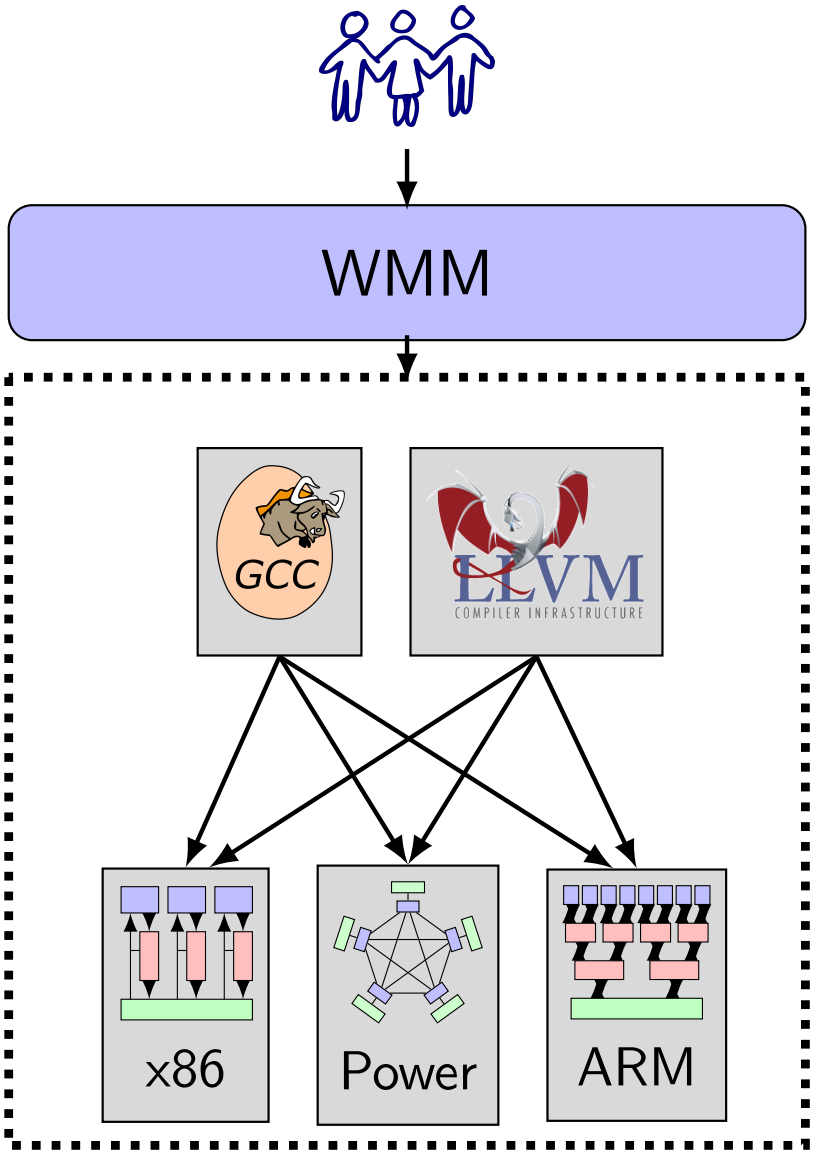
Overview



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 :-)

Overview



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

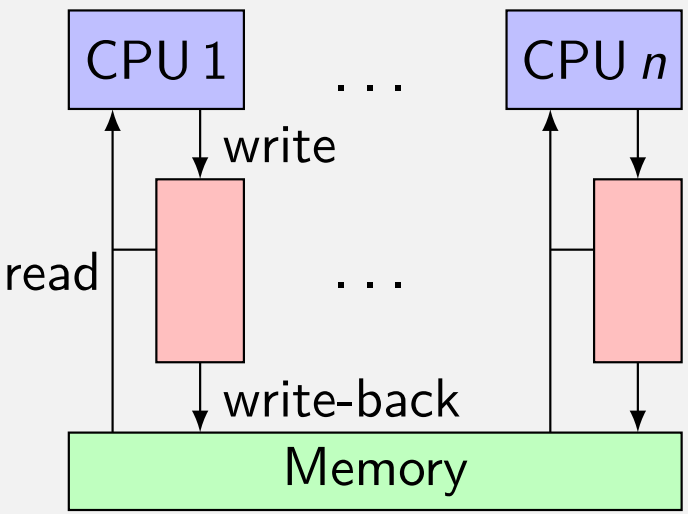
- Define the model as a set of consistency constraints on program *executions*.

Operational approach

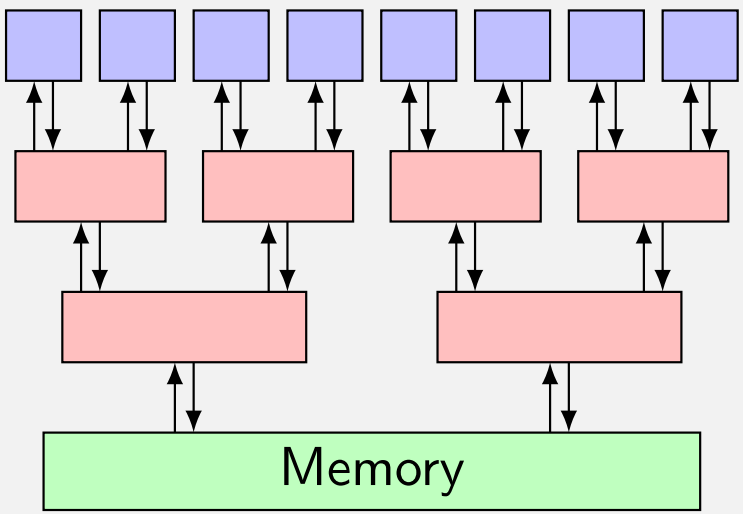
Define program semantics with an abstract machine.

- Works well for most hardware models.
- Very low-level $\rightsquigarrow$ 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

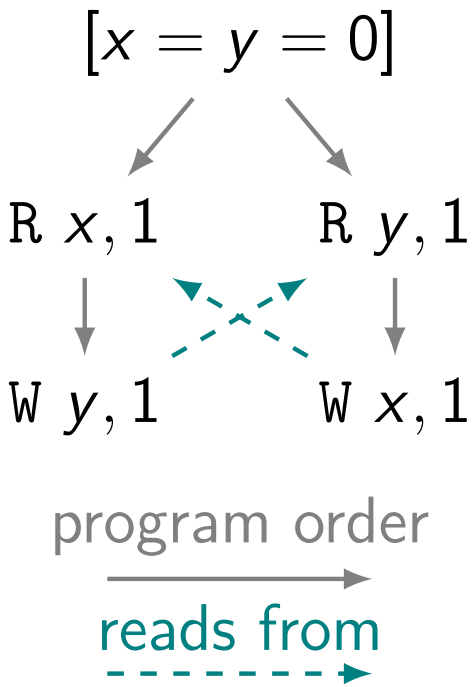
$$\begin{array}{l} a = x; \ \text{// 1} \\ x = 1; \end{array} \Big\| \quad y = x; \quad \Big\| \quad x = y;$$

Axiomatic approach

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

Example: Load-buffering

$$\begin{array}{l} a = x; \ \text{// 1} \\ y = 1; \end{array} \quad \Big\| \quad \begin{array}{l} b = y; \ \text{// 1} \\ x = b; \end{array}$$



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

The C11 memory model

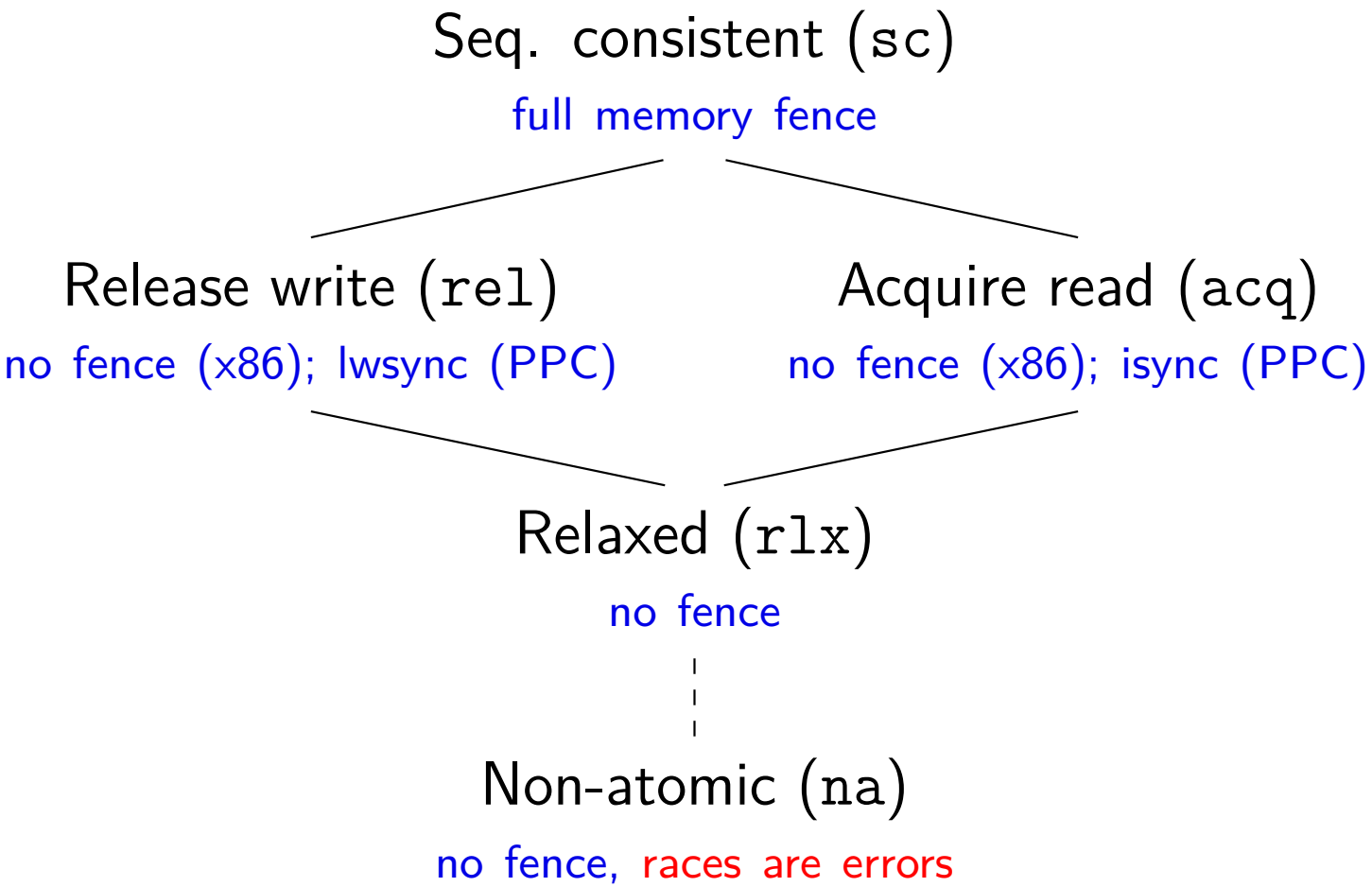
- Introduced by the ISO C/C++ 2011 standards.
- Defines the semantics of concurrent memory accesses.
- Adopted by the LLVM IR with some changes.
  (The differences are not relevant for this talk.)

The C11 memory model: Atomics

Two types of locations

Ordinary (Non-Atomic)	**Atomic**
Races are errors	Welcome to the expert mode

A spectrum of accesses



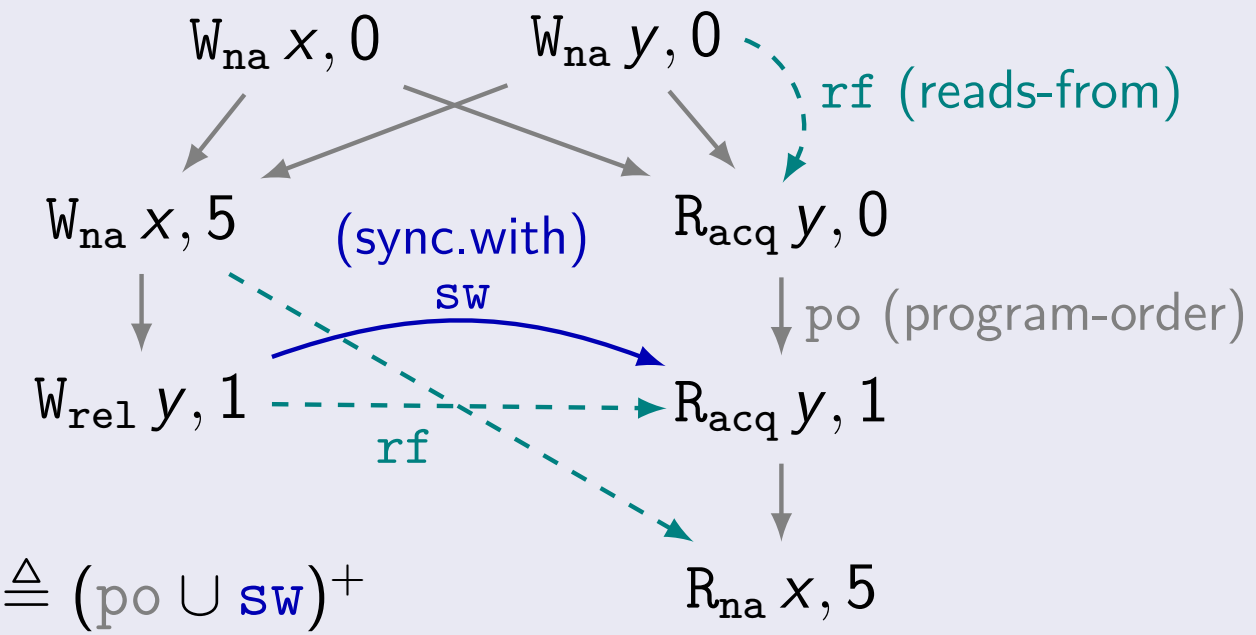
Explicit primitives for fences

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

Initially $x = y = 0$.

$$x = 5; \quad y.\text{store}(1, \textit{release}); \quad \| \quad \textbf{while}\ (y.\text{load}(\textit{acq}) == 0); \quad \text{print}(x);$$

One possible execution



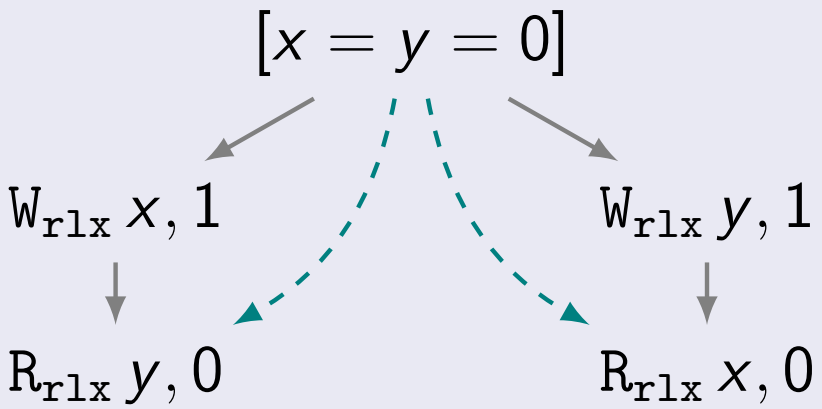
Relaxed behavior: store buffering

Initially $x = y = 0$.

$$\begin{array}{l} x.\mathsf{store}(1, \mathit{rlx}); \\ a = y.\mathsf{load}(\mathit{rlx}); \end{array} \quad \Big\| \quad \begin{array}{l} y.\mathsf{store}(1, \mathit{rlx}); \\ b = x.\mathsf{load}(\mathit{rlx}); \end{array}$$

This can return $a = b = 0$.

Justification



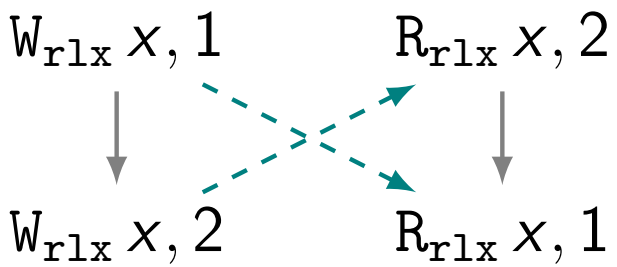
Behavior observed on x86/Power/ARM

Coherence

Programs with a single shared variable behave as under SC.

$$\begin{array}{l} x.\mathsf{store}(1, \mathit{rlx}); \\ x.\mathsf{store}(2, \mathit{rlx}); \end{array} \;\Big\|\; \begin{array}{l} a = x.\mathsf{load}(\mathit{rlx}); \\ b = x.\mathsf{load}(\mathit{rlx}); \end{array}$$

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

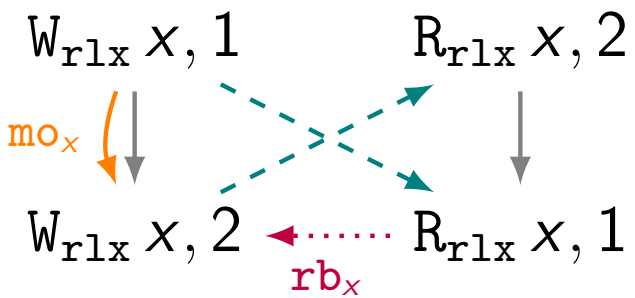


Coherence

Programs with a single shared variable behave as under SC.

$$\begin{array}{l} x.\mathsf{store}(1, \mathit{rlx}); \\ x.\mathsf{store}(2, \mathit{rlx}); \end{array} \quad \Big\| \quad \begin{array}{l} a = x.\mathsf{load}(\mathit{rlx}); \\ b = x.\mathsf{load}(\mathit{rlx}); \end{array}$$

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



- Modification order, $\mathtt{mo}_x$, total order of writes to x.
- Reads-before : $\mathtt{rb}_x \triangleq (\mathtt{rf}^{-1}; \mathtt{mo}_x) \cap (\neq)$
- Coherence : $\mathtt{hb} \cup \mathtt{rf}_x \cup \mathtt{mo}_x \cup \mathtt{rb}_x$ is acyclic for all x.

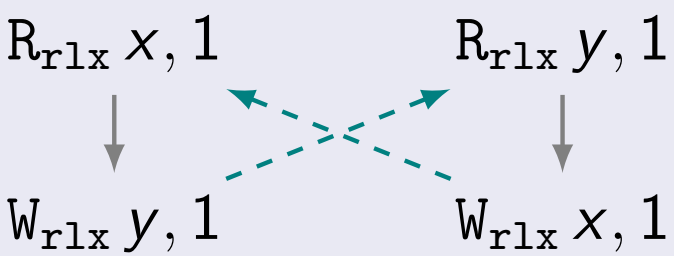
Causality cycles with relaxed accesses

Initially $x = y = 0$.

$$\begin{array}{l} \textbf{if } (x.load(rlx) == 1) \\ \quad y.store(1, rlx); \end{array} \quad \Big\| \quad \begin{array}{l} \textbf{if } (y.load(rlx) == 1) \\ \quad x.store(1, rlx); \end{array}$$

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

Justification



Relaxed accesses don't synchronize

No causality cycles with non-atomics

Initially $x = y = 0$.

$$\begin{array}{r} \textbf{if } (x == 1) \\ y = 1; \end{array} \quad \Big\| \quad \begin{array}{r} \textbf{if } (y == 1) \\ x = 1; \end{array}$$

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

Justification

Non-atomic read axiom:

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

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?

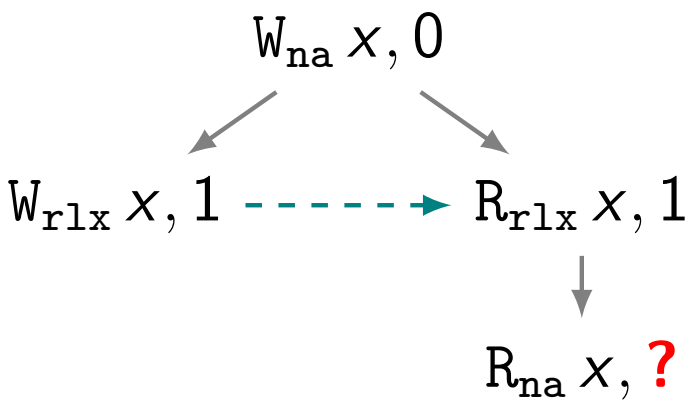
   ✗ No, it disallows intended program transformations.

Non-atomic reads of atomic variables are unsound!

Initially, $x = 0$.

$$x.\mathsf{store}(1, \mathit{rlx}); \;\Big\|\; \begin{array}{l} \textbf{if } (x.\mathsf{load}(\mathit{rlx}) == 1) \\ \quad t = (\mathsf{int})\, x; \end{array}$$

The program can get stuck!



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

Sequentialization is invalid

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

$$a = 1; \;\Big\|\; \textbf{if } (x.\text{load}(rlx) == 1) \;\; \textbf{if } (a == 1) \;\; y.\text{store}(1, rlx); \;\Big\|\; \textbf{if } (y.\text{load}(rlx) == 1) \;\; x.\text{store}(1, rlx);$$

The only possible output is:

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

Recall the non-atomic read axiom:

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

Tentative fixes

Remove non-atomic read axiom.

- gives extremely weak guarantees, if any

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

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

Related to the OOTA problem...

- More in a couple of slides

Monotonicity

"Adding synchronization should not introduce new behaviors"

Examples:

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

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

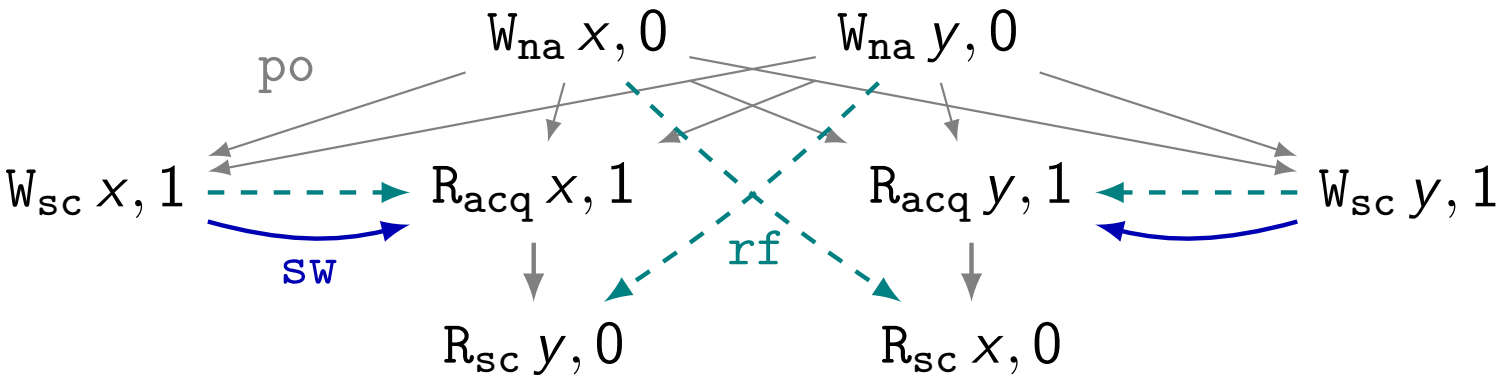
- Adding a memory fence
- Strengthening the access mode of an operation
- (Roach motel reorderings)

C11 semantics for SC accesses is broken! (PLDI'17)

IRIW-acq-sc

$$x_{\mathtt{sc}} = 1; \;\Big\|\; \begin{array}{l} a = x_{\mathtt{acq}}; \;\; /\!/\, 1 \\ c = y_{\mathtt{sc}}; \;\; /\!/\, 0 \end{array} \;\Big\|\; \begin{array}{l} b = y_{\mathtt{acq}}; \;\; /\!/\, 1 \\ d = x_{\mathtt{sc}}; \;\; /\!/\, 0 \end{array} \;\Big\|\; y_{\mathtt{sc}} = 1;$$

- C11 disallows the annotated behavior:



- The behavior is, however, allowed on POWER/ARM following the "trailing sync" compilation scheme.

Other problems fixed (POPL'15, POPL'16)

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 $(\texttt{po} \cup \texttt{rf})$-final events is unsound.

The OOTA problem

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

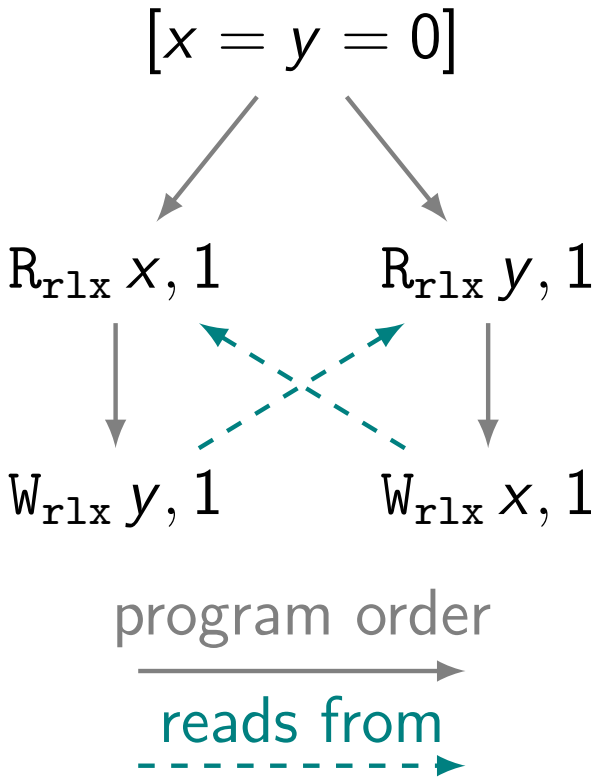
- Initially, $x = y = 0$.
- All accesses are "relaxed".

Load-buffering

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

This behavior must be allowed:

Power/ARM allow it

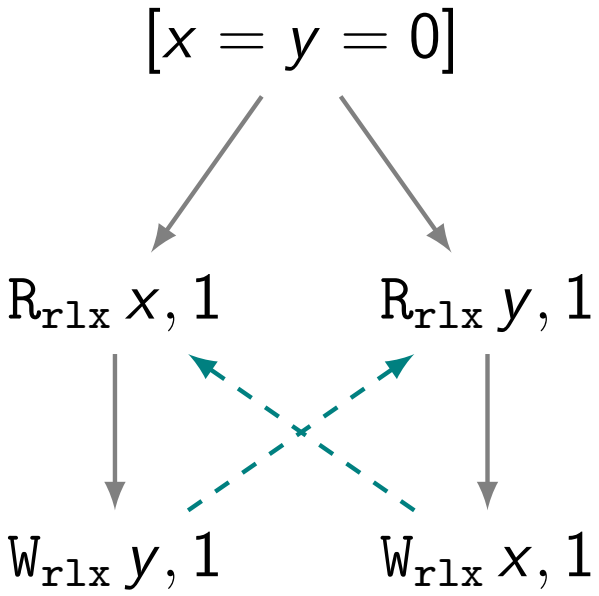


The *out-of-thin-air* problem in C11

Load-buffering + data dependency

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

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

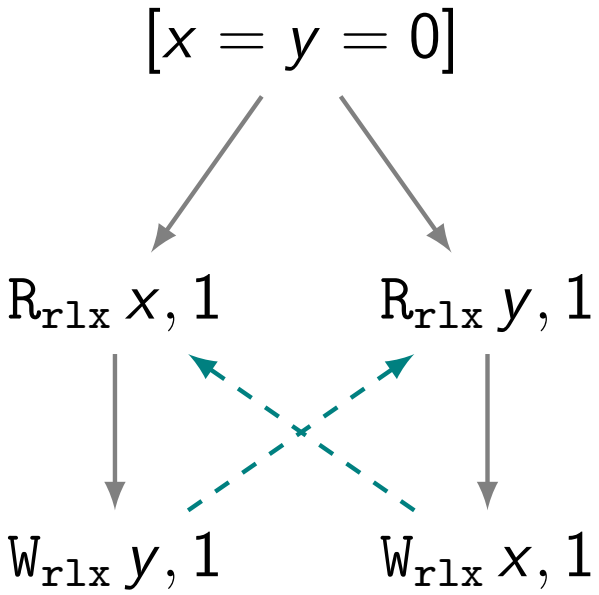
$$\begin{array}{l} a = x; \;\; /\!/\, 1 \\ y = a; \end{array} \;\Big\|\; x = y;$$

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

Load-buffering + control dependencies

$$\begin{array}{l} a = x; \;\; /\!/\, 1 \\ \texttt{if}\,(a == 1) \\ \quad y = 1; \end{array} \;\Big\|\; \begin{array}{l} \texttt{if}\,(y == 1) \\ \quad x = 1; \end{array}$$

The behavior should be forbidden:
DRF guarantee is broken!



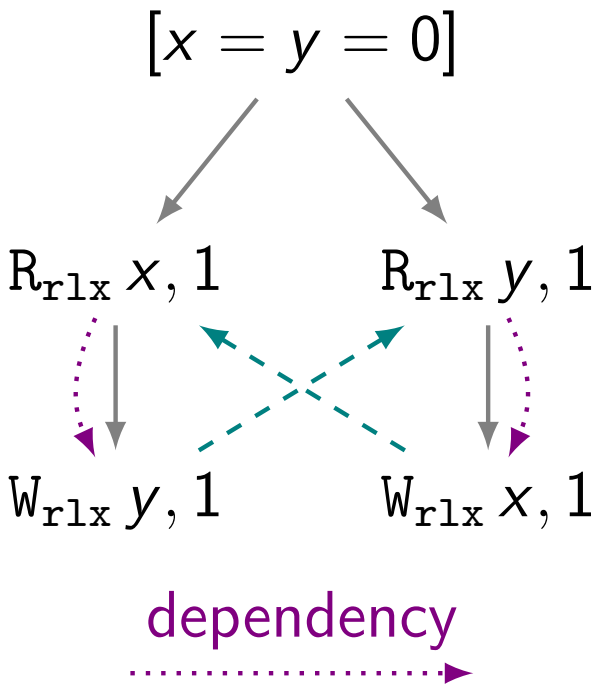
Same execution as before!
C11 allows these behaviors

The hardware solution

Keep track of syntactic dependencies, and forbid "dependency cycles".

Load-buffering + data dependency

$$a = x; \ // \ 1 \quad \| \quad x = y;$$
$$y = a;$$



The hardware solution

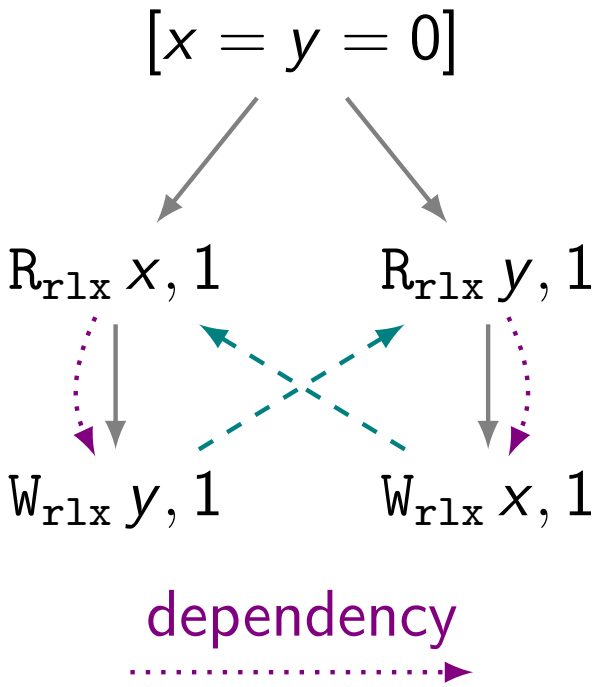
Keep track of syntactic dependencies,
and forbid "dependency cycles".

Load-buffering + data dependency

$$\begin{array}{l} a = x; \ \ /\!/\ 1 \\ y = a; \end{array} \quad \Big\| \quad x = y;$$

Load-buffering + fake dependency

$$\begin{array}{l} a = x; \ \ /\!/\ 1 \\ y = a + 1 - a; \end{array} \quad \Big\| \quad x = y;$$



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

$$x = y = 0$$

$$\begin{array}{l|l} x = 1; & y = 1; \\ a = y; \ // \ 0 & b = x; \ // \ 0 \end{array}$$

Simple operational semantics for C11's relaxed accesses

Store-buffering

$x = y = 0$

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

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

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

$$\langle \mathit{location} : \mathit{value} @ \mathit{timestamp} \rangle$$

- 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

$$x = y = 0$$

$x = 1;$	$\parallel$	▶ $y = 1;$
▶ $a = y;$ // 0	$\parallel$	$b = x;$ // 0

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$

T_1's view

x	y
~~0~~ 1	0

T_2's view

x	y
0	0

- Global memory is a pool of messages of the form

$$\langle \mathit{location} : \mathit{value} @ \mathit{timestamp} \rangle$$

- 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

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
~~0~~ 1	0

T_2's view

x	y
0	~~0~~ 1

- Global memory is a pool of messages of the form

$$\langle \mathit{location} : \mathit{value} @ \mathit{timestamp} \rangle$$

- 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

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	~~0~~
	1

- Global memory is a pool of messages of the form

$$\langle \mathit{location} : \mathit{value} @ \mathit{timestamp} \rangle$$

- 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

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	~~0~~
	1

- Global memory is a pool of messages of the form

$$\langle \mathit{location} : \mathit{value} @ \mathit{timestamp} \rangle$$

- 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

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	~~0~~
	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

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	~~0~~
	1

Coherence Test

$$x = 0$$

▶ $x = 1;$	▶ $x = 2;$
$a = x;$ // 2	$b = x;$ // 1

Memory

$\langle x : 0@0 \rangle$

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

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	~~0~~
	1

Coherence Test

$$x = 0$$

$x = 1;$	▶ $x = 2;$
▶ $a = x;$ // 2	$b = x;$ // 1

Memory

$\langle x : 0@0 \rangle$
$\langle x : 1@1 \rangle$

T_1's view

x
~~0~~
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

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	~~0~~
	1

Coherence Test

$$x = 0$$

$x = 1;$	$x = 2;$
▶ $a = x;$ // 2	▶ $b = x;$ // 1

Memory

$\langle x : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle x : 2@2 \rangle$

T_1's view

x
~~0~~
1

T_2's view

x
~~0~~
2

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

$x = 1;$	$\Vert$	$y = 1;$
$a = y;$ // 0	$\Vert$	$b = x;$ // 0
▶	$\Vert$	▶

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	~~0~~
	1

Coherence Test

$$x = 0$$

$x = 1;$	$\Vert$	$x = 2;$
$a = x;$ // 2	$\Vert$	▶ $b = x;$ // 1
▶	$\Vert$	

Memory

$\langle x : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle x : 2@2 \rangle$

T_1's view

x
~~0~~
~~1~~
2

T_2's view

x
~~0~~
2

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

$x = 1;$	$\Vert$	$y = 1;$
$a = y;$ // 0	$\Vert$	$b = x;$ // 0
▶	$\Vert$	▶

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle y : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	~~0~~
	1

Coherence Test

$$x = 0$$

$x = 1;$	$\Vert$	$x = 2;$
$a = x;$ // 2	$\Vert$	$b = x;$ // 1
▶	$\Vert$	▶

Memory

$\langle x : 0@0 \rangle$
$\langle x : 1@1 \rangle$
$\langle x : 2@2 \rangle$

T_1's view

x
~~0~~
~~1~~
2

T_2's view

x
~~0~~
2

Promises

Load-buffering

$$x = y = 0$$

$a = x;$ // 1	$\|\|$	$x = y;$
$y = 1;$		

- 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

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

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 $y = 1$;	▶ $x = y$;

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 $y = 1$;	▶ $x = y$;

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~~
	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

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@1 \rangle$

$\langle x : 1@1 \rangle$

T_1's view

x	y
0	0

T_2's view

x	y
~~0~~	~~0~~
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	$x = y;$
▶ $y = 1;$	▶

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@1 \rangle$

$\langle x : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
~~0~~	~~0~~
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	$x = y;$
$y = 1;$	

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@1 \rangle$

$\langle x : 1@1 \rangle$

T_1's view

x	y
~~0~~	~~0~~
1	1

T_2's view

x	y
~~0~~	~~0~~
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

$$x = y = 0$$

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

Load-buffering + dependency

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

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@1 \rangle$

$\langle x : 1@1 \rangle$

T_1's view

x	y
~~0~~	~~0~~
1	1

T_2's view

x	y
~~0~~	~~0~~
1	1

Must not admit the same execution!

Promises

Load-buffering

$$x = y = 0$$

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

Load-buffering + dependency

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

Key Idea

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

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.

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

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

Load buffering + fake dependency

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

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

Load buffering + dependency

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

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

Quiz #2

Is this behavior possible?

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

Quiz #2

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?

$$\begin{array}{l} a = x; \;\; /\!/\, 1 \\ x = 1; \end{array} \;\Big\|\; y = x; \;\Big\|\; x = y;$$

Quiz #3

Is this behavior possible?

$$\begin{array}{l} a = x; \;\; /\!/\, 1 \\ x = 1; \end{array} \;\Big\|\; y = x; \;\Big\|\; x = y;$$

Yes. And the ARM model allows it!

Quiz #3

Is this behavior possible?

$$\begin{array}{l} a = x; \;\; /\!/\, 1 \\ x = 1; \end{array} \;\Big\|\; y = x; \;\Big\|\; x = y;$$

Yes. And the ARM model allows it!

This behavior can be also explained by sequentialization:

$$\begin{array}{l} a = x; \;\; /\!/\, 1 \\ x = 1; \end{array} \Big\| y = x; \Big\| x = y; \quad \rightsquigarrow \quad \begin{array}{l} a = x; \;\; /\!/\, 1 \\ x = 1; \\ y = x; \end{array} \Big\| x = y;$$

Quiz #3

But, note that sequentialization is generally unsound in our model:

$$\begin{array}{l} a := x; \;\; /\!/ 1 \\ \textbf{if } a = 0 \textbf{ then} \\ \quad x := 1; \end{array} \;\Big\|\; y := x; \;\Big\|\; x := y; \quad \rightsquigarrow \quad \begin{array}{l} a := x; \;\; /\!/ 1 \\ \textbf{if } a = 0 \textbf{ then} \\ \quad x := 1; \\ y := x; \end{array} \;\Big\|\; x := y;$$

Taken from Viktor's lecture slides

The full model

- Atomic updates (e.g., CAS, fetch-and-add)
- Release/acquire fences and accesses
- Release sequences
- SC fences (no SC accesses)
- Plain accesses (C11's non-atomics & Java's normal accesses)

To achieve all of this we enrich our timestamps, messages, and thread views.

Taken from Viktor's lecture slides

Release/acquire accesses

Message-passing

$$
\begin{array}{c}
x = y = 0 \\
\begin{array}{l} x := 1; \\ y :=_{\mathbf{rel}} 1; \end{array} \quad \Big\| \quad \begin{array}{l} a := y_{\mathbf{acq}}; \quad /\!/ 1 \\ b := x; \quad /\!/ 1 \end{array}
\end{array}
$$

Taken from Viktor's lecture slides

Release/acquire accesses

Message-passing

$x = y = 0$

▶ $x := 1;$	▶ $a := y_{\textbf{acq}};$ // 1
$y :=_{\textbf{rel}} 1;$	$b := x;$ // 1

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

T_1's view

x	y
0	0

T_2's view

x	y
0	0

Taken from Viktor's lecture slides

Release/acquire accesses

Message-passing

$x = y = 0$

$x := 1;$	$\parallel$	▶ $a := y_{\textbf{acq}};$ // 1
▶ $y :=_{\textbf{rel}} 1;$		$b := x;$ // 1

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle x : 1@1 \rangle$

T_1's view

x	y
~~0~~	0
1	

T_2's view

x	y
0	0

Taken from Viktor's lecture slides

Release/acquire accesses

Message-passing

$$x = y = 0$$

$$\begin{array}{l|l} x := 1; & \blacktriangleright\ a := y_{\mathbf{acq}}; \quad /\!/ 1 \\ y :=_{\mathbf{rel}} 1; & \quad\ b := x; \quad /\!/ 1 \\ \blacktriangleright & \end{array}$$

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle x : 1@1 \rangle$

$\langle y : 1@1 \quad x@1 \rangle$

T_1's view

x	y
~~0~~	~~0~~
1	1

T_2's view

x	y
0	0

Taken from Viktor's lecture slides

Release/acquire accesses

Message-passing

$x = y = 0$

$x := 1;$	$a := y_{\textbf{acq}};$ // 1
$y :=_{\textbf{rel}} 1;$	▶ $b := x;$ // 1
▶	

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle x : 1@1 \rangle$

$\langle y : 1@1 \quad x@1 \rangle$

T_1's view

x	y
~~0~~	~~0~~
1	1

T_2's view

x	y
~~0~~	~~0~~
1	1

Taken from Viktor's lecture slides

Release/acquire accesses

Message-passing

$x = y = 0$

$x := 1;$	$a := y_{\textbf{acq}};$ // 1
$y :=_{\textbf{rel}} 1;$	$b := x;$ // 1
▶	▶

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle x : 1@1 \rangle$

$\langle y : 1@1 \quad x@1 \rangle$

T_1's view

x	y
~~0~~	~~0~~
1	1

T_2's view

x	y
~~0~~	~~0~~
1	1

Taken from Viktor's lecture slides

Certification is needed at every step

Key lemma for DRF

Races only on RA under promise-free semantics
$\Rightarrow$ only promise-free behaviors

$$
\begin{array}{l||l||l}
w :=_{\mathbf{rel}} 1; & \begin{array}{l} \textbf{if } w_{\mathbf{acq}} = 1 \textbf{ then} \\ \quad z := 1; \\ \textbf{else} \\ \quad y :=_{\mathbf{rel}} 1; \\ a := x; \quad /\!/\, 1 \\ \textbf{if } a = 1 \textbf{ then} \\ \quad z := 1; \end{array} & \begin{array}{l} \textbf{if } y_{\mathbf{acq}} = 1 \textbf{ then} \\ \quad \textbf{if } z = 1 \textbf{ then} \\ \quad\quad x := 1; \end{array}
\end{array}
$$

Taken from Viktor's lecture slides

The full model (POPL'17)

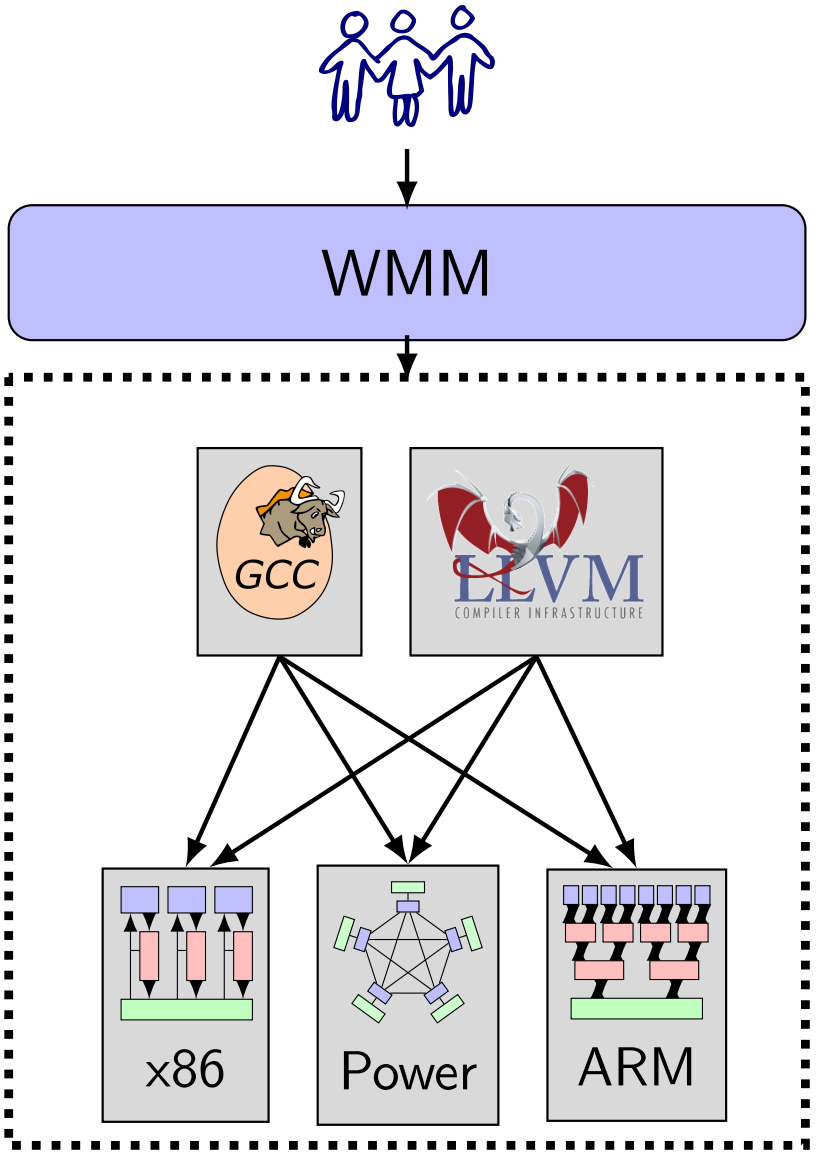
We have extended this basic idea to handle:

- 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 **OOTA** requires an entirely different approach.
- The **promising model** may be the solution.

Initially, $X = Y = 0$.

$X = 2;$	$\|\|$	$X = 1;$
$a = X;$ // 3		$b = X;$ // 2
if $(a \neq 2)$		$c = Y;$ // 1
$\quad Y = 1;$		**if** (c)
		$\quad X = 3;$

(Coh-CYC)

This example is taken from the paper "Grounding Thin-Air Reads with Event Structures" by Soham Chakraborty and Viktor Vafeiadis (POPL 2019).