

Weak Memory Concurrency in C/C++11

Ori Lahav



Haifa::C++ meeting
November 21, 2017

Example: Dekker's mutual exclusion

Initially, $x = y = 0$.

$x := 1$;

$a := y$;

if ($a = 0$) **then**

/ critical section */*

||

$y := 1$;

$b := x$;

if ($b = 0$) **then**

/ critical section */*

Example: Dekker's mutual exclusion

Initially, $x = y = 0$.

$x := 1$;

$a := y$;

if ($a = 0$) **then**

/ critical section */*

||

$y := 1$;

$b := x$;

if ($b = 0$) **then**

/ critical section */*

Is it safe?

Example: Dekker's mutual exclusion

Initially, $x = y = 0$.

```
x := 1;
```

```
a := y; // 0
```

```
if (a = 0) then
```

```
    /* critical section */
```

```
y := 1;
```

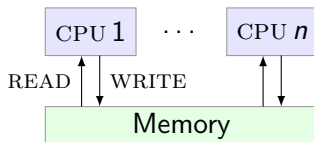
```
b := x; // 0
```

```
if (b = 0) then
```

```
    /* critical section */
```

Is it safe?

Yes, if we assume sequential consistency (SC):



Example: Dekker's mutual exclusion

Initially, $x = y = 0$.

```
x := 1;
```

```
a := y; // 0
```

```
if (a = 0) then
```

```
    /* critical section */
```

```
y := 1;
```

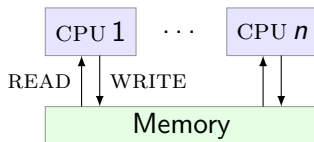
```
b := x; // 0
```

```
if (b = 0) then
```

```
    /* critical section */
```

Is it safe?

Yes, if we assume sequential consistency (SC):



No existing hardware implements SC!

- ▶ SC is very expensive (memory ~ 100 times slower than CPU).
- ▶ SC does not scale to many processors.

Example: Shared-memory concurrency in C++

```
int X, Y, a, b;

void thread1() {
    X = 1;
    a = Y;
}

void thread2() {
    Y = 1;
    b = X;
}
```

```
int main () {
    int cnt = 0;

    do {
        X = 0; Y = 0;

        thread first(thread1);
        thread second(thread2);

        first.join();
        second.join();
        cnt++;

    } while (a != 0 || b != 0);

    printf("%d\n", cnt);
    return 0;
}
```

Example: Shared-memory concurrency in C++

```
int X, Y, a, b;

void thread1() {
    X = 1;
    a = Y;
}

void thread2() {
    Y = 1;
    b = X;
}
```

If Dekker's mutual exclusion is safe, this program will not terminate

```
int main () {
    int cnt = 0;

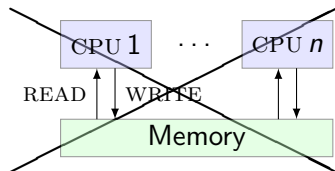
    do {
        X = 0; Y = 0;

        thread first(thread1);
        thread second(thread2);

        first.join();
        second.join();
        cnt++;
    } while (a != 0 || b != 0);

    printf("%d\n", cnt);
    return 0;
}
```

Weak memory models



We look for a *substitute for SC*:

Unambiguous specification

- ▶ What are the possible outcomes of a multithreaded program?

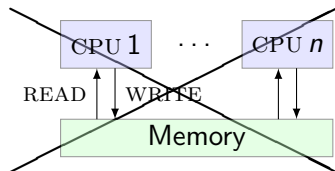
Typically called a **weak memory model (WMM)**

- ▶ Allows more behaviors than SC.

Amenable to formal reasoning

- ▶ Can prove theorems about the model.

Weak memory models



We look for a *substitute for SC*:

Unambiguous specification

- ▶ What are the possible outcomes of a multithreaded program?

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

- ▶ Allows more behaviors than SC.

Amenable to formal reasoning

- ▶ Can prove theorems about the model.

But it is not easy to get right

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

The Problem of Programming Language Concurrency Semantics

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod,
and Peter Sewell

University of Cambridge

“Disturbingly, 40+ years after the first relaxed-memory hardware was introduced (the IBM 370/158MP), the field still *does not have a credible proposal for the concurrency semantics* of any general-purpose high-level language that includes high performance shared-memory concurrency primitives. This is a *major open problem* for programming language semantics.”

European Symposium on Programming (ESOP) 2015

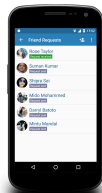
Plan for rest of the talk

1. Challenges for memory models
2. The C/C++11 memory model
3. The “out-of-thin-air” problem
4. A solution: a promising semantics

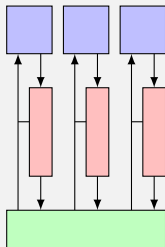
Plan for rest of the talk


1. Challenges for memory models
2. The C/C++11 memory model
3. The “out-of-thin-air” problem
4. A solution: a promising semantics

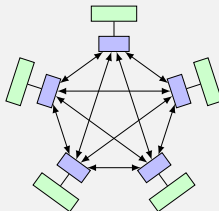
Challenge 1: Various hardware models




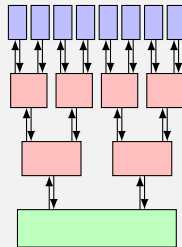
x86-TSO  
(2010)



POWER 
(2011)



ARMv8 
(2016)



Store buffering in x86-TSO



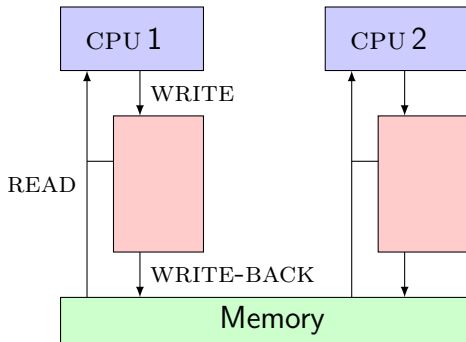
Initially, $x = y = 0$.

$x := 1;$

$y := 1;$

$a := y; \text{ // } 0$

$b := x; \text{ // } 0$



Store buffering in x86-TSO



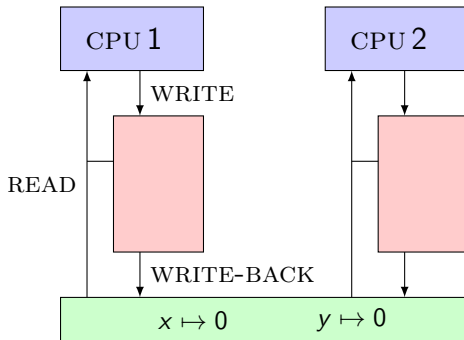
Initially, $x = y = 0$.

► $x := 1$;

$a := y$; // 0

► $y := 1$;

$b := x$; // 0



Store buffering in x86-TSO



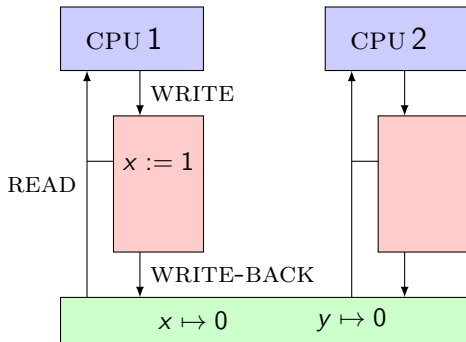
Initially, $x = y = 0$.

$x := 1;$

► $y := 1;$

► $a := y; \text{ // } 0$

$b := x; \text{ // } 0$



Store buffering in x86-TSO



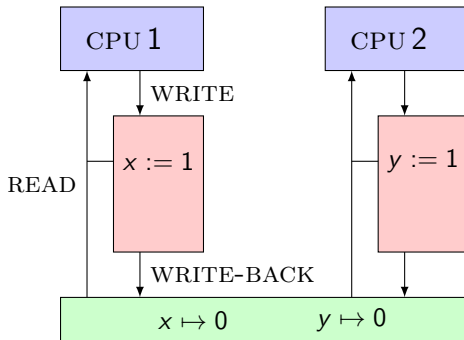
Initially, $x = y = 0$.

$x := 1;$

$y := 1;$

► $a := y; \text{ // } 0$

► $b := x; \text{ // } 0$



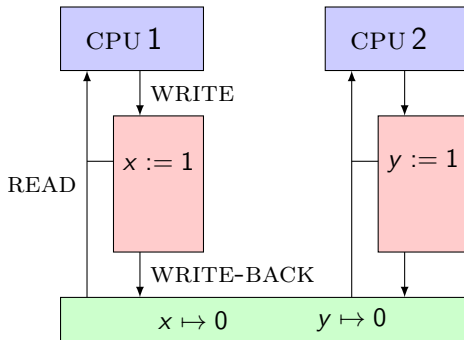
Store buffering in x86-TSO



Initially, $x = y = 0$.

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

```
y := 1;  
fence;  
b := x; // 0
```

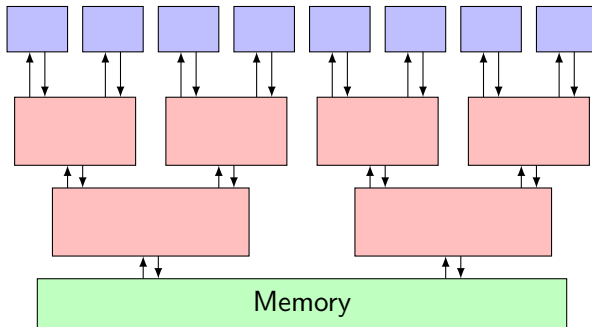


Load buffering in ARM



Initially, $x = y = 0$.

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := 1;$	\parallel	$x := b;$

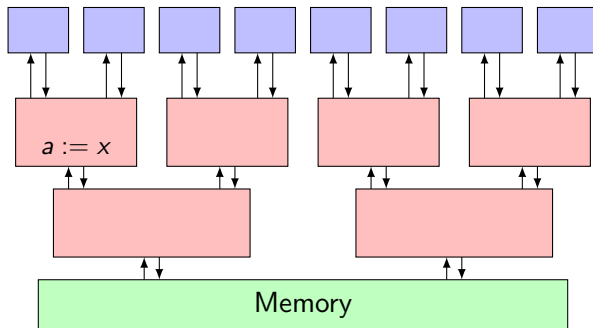


Load buffering in ARM



Initially, $x = y = 0$.

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := 1;$	\parallel	$x := b;$

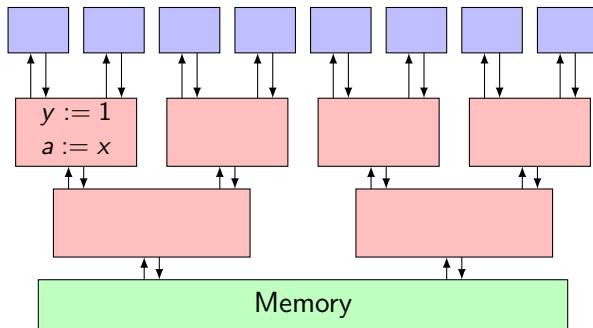


Load buffering in ARM



Initially, $x = y = 0$.

$a := x; \text{ // } 1$		$b := y; \text{ // } 1$
$y := 1;$		$x := b;$

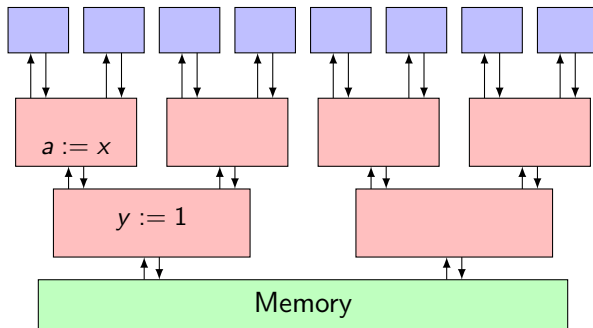


Load buffering in ARM



Initially, $x = y = 0$.

$a := x; \text{ // } 1$		$b := y; \text{ // } 1$
$y := 1;$		$x := b;$

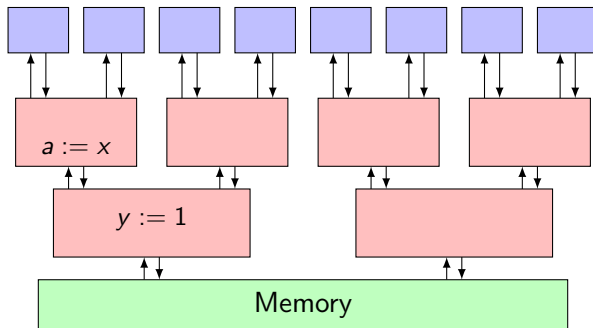


Load buffering in ARM



Initially, $x = y = 0$.

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



Challenge 2: Compilers stir the pot

Initially, $x = y = 0$.

$$\begin{array}{l} x := 1; \\ y := 1; \end{array} \parallel \begin{array}{l} a := x; \\ b := y; \text{ // } 1 \\ c := x; \text{ // } 0 \end{array}$$

X forbidden under SC

Challenge 2: Compilers stir the pot

Initially, $x = y = 0$.

$x := 1;$ \parallel $a := x;$
 $y := 1;$ \parallel $b := y; \text{ // } 1$
 \parallel $c := x; \text{ // } 0$

X forbidden under SC



*compiler
optimization*



$x := 1;$ \parallel $a := x;$
 $y := 1;$ \parallel $b := y; \text{ // } 1$
 \parallel $c := a; \text{ // } 0$

✓ allowed under SC

Challenge 3: Transformations do not suffice

Program transformations fail short to explain some weak behaviors:

Message passing (MP)

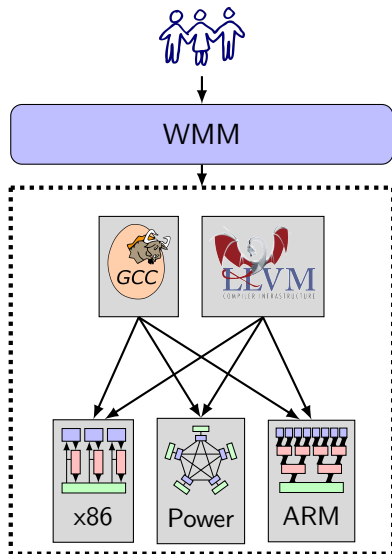
$$\begin{array}{l} x := 1; \\ y := 1; \end{array} \parallel \begin{array}{l} a := y; \text{ // } 1 \\ b := x; \text{ // } 0 \end{array}$$

Independent reads of independent writes (IRIW)

$$\begin{array}{l} a := x; \text{ // } 1 \\ b := y; \text{ // } 0 \end{array} \parallel x := 1; \parallel y := 1; \parallel \begin{array}{l} c := y; \text{ // } 1 \\ d := x; \text{ // } 0 \end{array}$$

ARM-weak

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



WMM desiderata

1. Formal and comprehensive
2. Not too weak
(good for programmers)
3. Not too strong
(good for hardware)
4. Admits optimizations
(good for compilers)

The C11 memory model

- ▶ Introduced by the ISO C/C++ 2011 standards.
- ▶ Defines the semantics of **concurrent** memory accesses.

The C11 memory model: Atomics

Two types of accesses

**Ordinary
(Non-Atomic)**

Races are **errors**

Atomic

Welcome to the
expert mode

The C11 memory model: Atomics

Two types of accesses

**Ordinary
(Non-Atomic)**

Races are **errors**

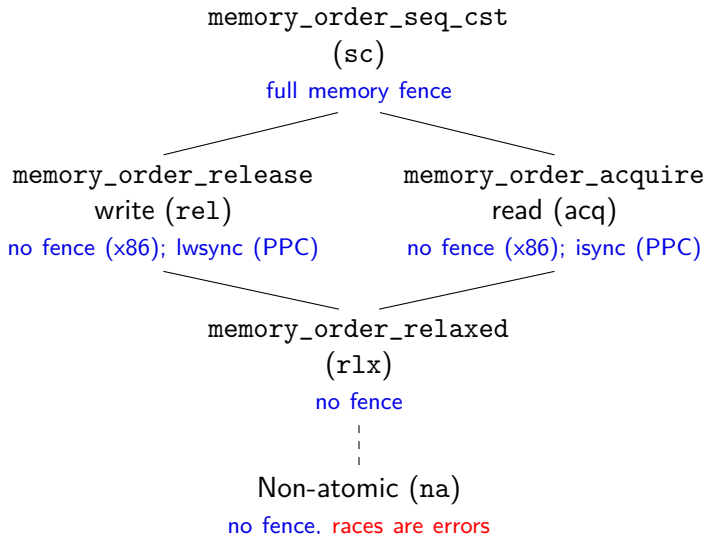
Atomic

Welcome to the
expert mode

DRF (data race freedom) guarantee

no data races
under SC \Rightarrow only
SC behaviors

A spectrum of access modes



+ Explicit primitives for fences

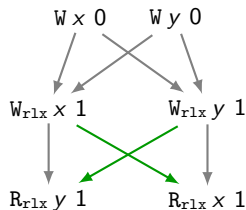
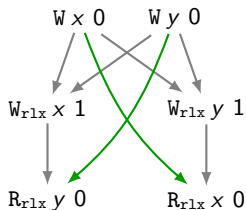
C11: a declarative memory model

Declarative semantics abstracts away from implementation details.

1. a program \leadsto a set of directed graphs (called: *execution graphs*)
2. The memory model defines what executions are *consistent*.
3. The semantics of a program is the set of its *consistent executions*.
4. C/C++11 also has *catch-fire* semantics (i.e., forbidden data races).

Execution graphs

Store buffering (SB)

$$\begin{array}{l} x = y = 0 \\ x :=_{rlx} 1 \parallel y :=_{rlx} 1 \\ a :=_{rlx} y \parallel b :=_{rlx} x \end{array}$$


Relations

- ▶ Program order, po
- ▶ Reads-from, rf

Figure 2. Semantics of closed program expressions

$$\begin{array}{ll}
\text{where } \text{iswrite}_{\ell, v}(a) \stackrel{\text{def}}{=} \exists X, \ell_{\text{old}}. \text{lab}(a) \in \{W_X(\ell, v), \text{RMW}_X(\ell, \ell_{\text{old}}, v)\} & \text{iswrite}_{\ell}(a) \stackrel{\text{def}}{=} \exists v. \text{iswrite}_{\ell, v}(a) \\
\text{isread}_{\ell, v}(a) \stackrel{\text{def}}{=} \exists X, \ell_{\text{new}}. \text{lab}(a) \in \{R_X(\ell, v), \text{RMW}_X(\ell, v, \ell_{\text{new}})\} & \text{etc.} \\
\text{rsElem}(a, b) \stackrel{\text{def}}{=} \text{sameThread}(a, b) \vee \text{isrmw}(b) \\
\text{rseq}(a) \stackrel{\text{def}}{=} \{a\} \cup \{b \mid \text{rsElem}(a, b) \wedge \text{mo}(a, b) \wedge (\forall c. \text{mo}(a, c) \wedge \text{mo}(c, b) \Rightarrow \text{rsElem}(a, c))\} \\
\text{sw} \stackrel{\text{def}}{=} \{(a, b) \mid \text{mode}(a) \in \{\text{rel}, \text{rel_acq}, \text{sc}\} \wedge \text{mode}(b) \in \{\text{acq}, \text{rel_acq}, \text{sc}\} \wedge \text{rf}(b) \in \text{rseq}(a)\} \\
\text{hb} \stackrel{\text{def}}{=} (\text{sb} \cup \text{sw})^+ \\
\text{hb}_{\ell} \stackrel{\text{def}}{=} \{(a, b) \in \text{hb} \mid \text{iswrite}_{\ell}(a) \wedge \text{iswrite}_{\ell}(b)\} \\
X_{\text{SeqCst}} \stackrel{\text{def}}{=} \{(a, b) \in X \mid \text{isSeqCst}(a) \wedge \text{isSeqCst}(b)\} \\
\text{isc}(a, b) \stackrel{\text{def}}{=} \text{iswrite}_{\text{lock}(b)}(a) \wedge \text{sc}(a, b) \wedge \nexists c. \text{sc}(a, c) \wedge \text{sc}(c, b) \wedge \text{iswrite}_{\text{lock}(b)}(c)
\end{array}$$

Figure 3. Axioms satisfied by consistent C11 executions, $\text{Consistent}(\mathcal{A}, \text{lab}, \text{sb}, \text{rf}, \text{mo}, \text{sc})$.

Figure 4. Sample executions violating coherency conditions (Batty et al. 2011).

Basic ingredients of execution graph consistency

1. SC-per-location (a.k.a. coherence)
2. Release/acquire synchronization
3. Global conditions on SC accesses

Basic ingredients of execution graph consistency

1. SC-per-location (a.k.a. coherence)
2. Release/acquire synchronization
3. Global conditions on SC accesses

SC-per-location

Definition (Declarative definition of SC)

G is *SC-consistent* if there exists a relation sc s.t. the following hold:

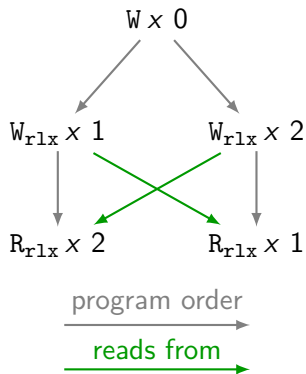
- ▶ sc is a total order on the events of G .
- ▶ If $po \cup rf \subseteq sc$.
- ▶ If $\langle a, b \rangle \in rf$ then there does not exist $c \in W_{loc(a)}$ such that $\langle a, c \rangle \in sc$ and $\langle c, b \rangle \in sc$.

Definition (SC-per-location)

G satisfies *SC-per-location* if for every location x , there exists a relation sc_x s.t. the following hold:

- ▶ sc_x is a total order on the events of G that access x .
- ▶ If $po \cup rf \subseteq sc_x$.
- ▶ If $\langle a, b \rangle \in rf$ then there does not exist $c \in W_x$ such that $\langle a, c \rangle \in sc_x$ and $\langle c, b \rangle \in sc_x$.

SC-per-location: Example

$$\begin{array}{l} x = 0 \\ x :=_{rlx} 1 \parallel x :=_{rlx} 2 \\ a := x_{rlx} \parallel b := x_{rlx} \end{array}$$


inconsistent!

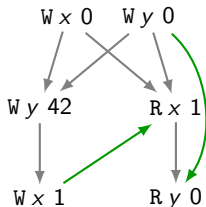
Release/acquire synchronization

SC-per-location is often too weak:

- It does not support the message passing idiom:

Message passing (MP)

```
y := 42; || a := x; // 1  
x := 1;   || b := y; // 0
```



- We cannot even implement locks:

Simple lock

```
lock(); || lock();  
x := 1; || a := x; // 1  
x := 2; || unlock();  
unlock();
```

Synchronization in C/C++11 through examples

```
int y = 0;  
int x = 0;  
y = 42; || if(x == 1){  
x = 1;    ||     print(y);  
         || }  
         ||
```


Synchronization in C/C++11 through examples

1

```
int y = 0;  
int x = 0;  
y = 42; || if(x == 1){  
x = 1;    ||     print(y);  
         || }  
         ||
```

Synchronization in C/C++11 through examples

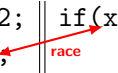
1

```
int y = 0;  
int x = 0;  
y = 42; || if(x == 1){  
x = 1;    race print(y);  
          }  
          }
```

Synchronization in C/C++11 through examples

1

```
int y = 0;
int x = 0;
y = 42; || if(x == 1){
x = 1;    ||     print(y);
          ||     }
          ||
          ||
```



2

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xrlx == 1){
x =rlx 1; ||     print(y);
          ||     }
          ||
```

Synchronization in C/C++11 through examples

1

```
int y = 0;
int x = 0;
y = 42; || if(x == 1){
x = 1;    race print(y);
          }

```

2

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xrlx == 1){
x =rlx 1; race print(y);
          }

```

Synchronization in C/C++11 through examples

1

```
int y = 0;
int x = 0;
y = 42; || if(x == 1){
x = 1;    race print(y);
          }
||
```

2

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xrlx == 1){
x =rlx 1; race print(y);
          }
||
```

3

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xacq == 1){
x =rel 1;    print(y);
          }
||
```

Synchronization in C/C++11 through examples

1

```
int y = 0;
int x = 0;
y = 42; || if(x == 1){
x = 1;    race print(y);
          }
||
```

2

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xrlx == 1){
x =rlx 1; race print(y);
          }
||
```

3

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xacq == 1){
x =rel 1; rf print(y);
          }
||
```

Synchronization in C/C++11 through examples

1

```
int y = 0;
int x = 0;
y = 42; || if(x == 1){
x = 1;    race print(y);
          }
||
```

2

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xrlx == 1){
x =rlx 1; race print(y);
          }
||
```

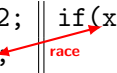
3

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xacq == 1){
x =rel 1; rf print(y);
          }
|| sw
```

Synchronization in C/C++11 through examples

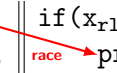
1

```
int y = 0;
int x = 0;
y = 42; || if(x == 1){
x = 1;    race print(y);
          }
```

A red arrow labeled "race" points from the assignment `x = 1;` to the condition `if(x == 1){` in the subsequent line, indicating a race condition where the value of `x` is not guaranteed to be updated before the condition is checked.

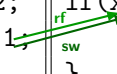
2

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xrlx == 1){
x =rlx 1;    race print(y);
              }
```

A red arrow labeled "race" points from the relaxed assignment `x =rlx 1;` to the relaxed read `if(xrlx == 1){` in the subsequent line, indicating a race condition.

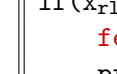
3

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xacq == 1){
x =rel 1;    rf      sw print(y);
              }
```

A green arrow labeled "rf" (read-after-write) points from the relaxed assignment `x =rel 1;` to the acquire-read `if(xacq == 1){` in the subsequent line. A green arrow labeled "sw" (store-write) points from the assignment `y = 42;` to the `print(y);` statement, indicating a write-after-write dependency.

4

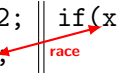
```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xrlx == 1){
fencerel;    fenceacq;
x =rlx 1;    print(y);
              }
```

The diagram shows two fences: `fencerel;` after the relaxed write `x =rlx 1;` and `fenceacq;` before the relaxed read `if(xrlx == 1){`. This ensures that the write to `x` is globally visible before the read is executed, despite the relaxed memory ordering.

Synchronization in C/C++11 through examples

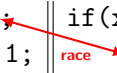
1

```
int y = 0;
int x = 0;
y = 42; || if(x == 1){
x = 1;    race print(y);
          }
```

A red arrow labeled "race" points from the assignment `x = 1;` on the left to the condition `if(x == 1){` on the right, indicating a race condition where the value of `x` is being updated and read simultaneously.

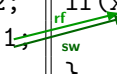
2

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xrlx == 1){
x =rlx 1;    race print(y);
              }
```

A red arrow labeled "race" points from the relaxed assignment `x =rlx 1;` on the left to the relaxed read `if(xrlx == 1){` on the right, indicating a race condition.

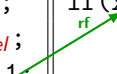
3

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xacq == 1){
x =rel 1;    rf      sw print(y);
              }
```

A green arrow labeled "rf" points from the relaxed write `x =rel 1;` on the left to the acquire read `if(xacq == 1){` on the right. A label "sw" is placed below the arrow, indicating a write-swaps relationship.

4

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xrlx == 1){
fencerel;    rf      fenceacq;
x =rlx 1;    print(y);
              }
```

A green arrow labeled "rf" points from the relaxed write `x =rlx 1;` on the left to the relaxed fence `fencerel;` on the right. A label "sw" is placed below the arrow, indicating a write-swaps relationship.

Synchronization in C/C++11 through examples

1

```
int y = 0;
int x = 0;
y = 42; || if(x == 1){
x = 1;    |  print(y);
          |  }
          |  
```

Diagram illustrating a race condition. A red arrow labeled "race" points from the assignment `x = 1;` to the condition `if(x == 1){` in the same thread, indicating that the thread's own update to `x` is not yet visible to its subsequent conditional check.

2

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xrlx == 1){
x =rlx 1; |  print(y);
          |  }
          |  
```

Diagram illustrating a race condition with relaxed memory ordering. A red arrow labeled "race" points from the assignment `x =rlx 1;` to the condition `if(xrlx == 1){` in the same thread, indicating that the thread's own update to `x` is not yet visible to its subsequent conditional check.

3

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xacq == 1){
x =rel 1; |  print(y);
          |  }
          |  
```

Diagram illustrating a synchronization point using acquire and release semantics. A green arrow labeled "rf" (read-after-write) points from the condition `if(xacq == 1){` to the assignment `x =rel 1;` in the same thread, indicating that the thread's own update to `x` is visible to its subsequent conditional check.

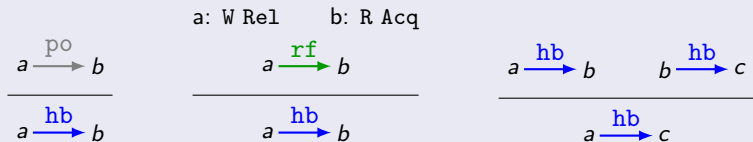
4

```
int y = 0;
atomic<int> x = 0;
y = 42; || if(xrlx == 1){
fencerel; |  fenceacq;
          |  print(y);
          |  }
          |  
```

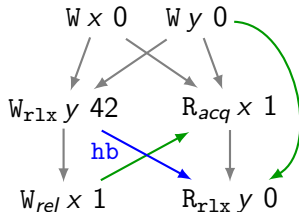
Diagram illustrating a synchronization point using fences. A green arrow labeled "rf" (read-after-write) points from the condition `if(xrlx == 1){` to the assignment `x =rlx 1;` in the same thread, indicating that the thread's own update to `x` is visible to its subsequent conditional check.

The “happens-before” relation

Definition (happens-before)



- ▶ **hb** should be acyclic.
- ▶ The SC-per-location orders should contain **hb**.
- ▶ Using acquire CAS's and release writes, we can implement locks.



SC accesses and fences

Store buffer

$$\begin{array}{l} x := 1; \\ a := y; \text{ // } 0 \end{array} \parallel \begin{array}{l} y := 1; \\ b := x; \text{ // } 0 \end{array}$$

How to guarantee only SC behaviors (*i.e.*, $a = 1 \vee b = 1$)?

$$\begin{array}{l} x :=_{\text{sc}} 1; \\ a := y_{\text{sc}}; \end{array} \parallel \begin{array}{l} y :=_{\text{sc}} 1; \\ b := x_{\text{sc}}; \end{array} \approx \begin{array}{l} x :=_{\text{rlx}} 1; \\ \mathbf{fence}_{\text{sc}}; \\ a := y_{\text{rlx}}; \end{array} \parallel \begin{array}{l} y :=_{\text{rlx}} 1; \\ \mathbf{fence}_{\text{sc}}; \\ b := x_{\text{rlx}}; \end{array}$$

SC semantics

- ▶ Perhaps surprisingly, the semantics of SC atomics is the *most complicated* part of the model.
- ▶ C/C++11 provides *too strong* semantics (a correctness problem!)

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

- ▶ In addition, its semantics for SC fences is *too weak*.

$$\begin{array}{l} a := x_{\text{acq}}; \text{ // } 1 \\ \text{fence-sc;} \\ b := y_{\text{acq}}; \text{ // } 0 \end{array} \parallel x :=_{\text{rel}} 1; \parallel y :=_{\text{rel}} 1; \parallel \begin{array}{l} c := y_{\text{acq}}; \text{ // } 1 \\ \text{fence-sc;} \\ d := x_{\text{acq}}; \text{ // } 0 \end{array}$$

- ▶ Recently, the standard committee fixed the specification following:
[Repairing Sequential Consistency in C/C++11 PLDI'17]

The “out-of-thin-air” problem

C/C++11 is too weak

non-atomic ☐ relaxed ☐ release/acquire ☐ sc

C/C++11 is too weak

non-atomic ☐ relaxed ☐ release/acquire ☐ sc

Load-buffering

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := 1;$	\parallel	$x := b;$

C/C++11 allows this behavior
because **POWER & ARM allow it!**

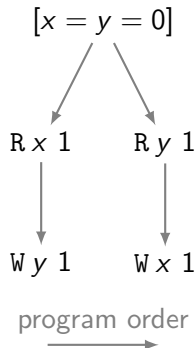
C/C++11 is too weak

non-atomic \sqsubset relaxed \sqsubset release/acquire \sqsubset sc

Load-buffering

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := 1;$		$x := b;$

C/C++11 allows this behavior
because **POWER & ARM allow it!**



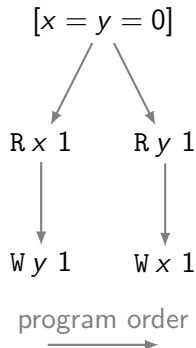
C/C++11 is too weak

non-atomic \sqsubset relaxed \sqsubset release/acquire \sqsubset sc

Load-buffering

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := 1;$		$x := b;$

C/C++11 allows this behavior
because **POWER & ARM allow it!**



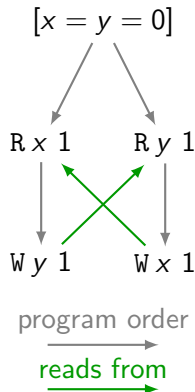
C/C++11 is too weak

non-atomic \square **relaxed** \square release/acquire \square sc

Load-buffering

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := 1;$		$x := b;$

C/C++11 allows this behavior
because **POWER & ARM allow it!**



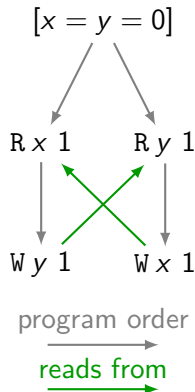
C/C++11 is too weak

non-atomic \square **relaxed** \square release/acquire \square sc

Load-buffering

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := 1;$		$x := b;$

C/C++11 allows this behavior
because **POWER & ARM allow it!**



C/C++11 is too weak

non-atomic \sqsubset **relaxed** \sqsubset release/acquire \sqsubset sc

Load-buffering

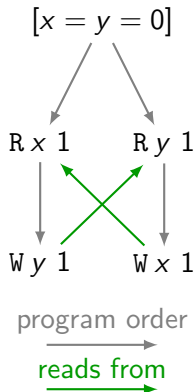
$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := 1;$		$x := b;$

C/C++11 allows this behavior
because **POWER & ARM allow it!**

Load-buffering + data dependency

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := a;$		$x := b;$

C/C++11 allows this behavior.
Values appear out-of-thin-air!
(no hardware/compiler exhibit this behavior)



C/C++11 is too weak

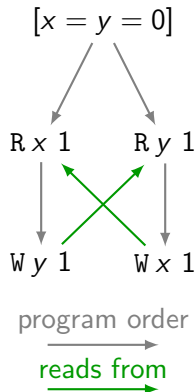
non-atomic \sqsubset relaxed \sqsubset release/acquire \sqsubset sc

Load-buffering + control dependency

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
if ($a = 1$)		if ($b = 1$)
$y := 1;$		$x := 1;$

C/C++11 allows this behavior.

The DRF guarantee is broken!



C/C++11 is too weak

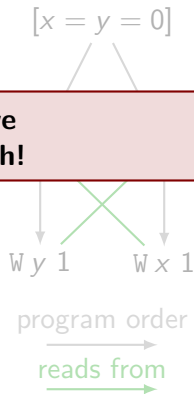
non-atomic ☐ relaxed ☐ release/acquire ☐ sc

Load-buffering + control dependency

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

**The three examples have
the same execution graph!**

The DRF guarantee is broken!



The hardware solution

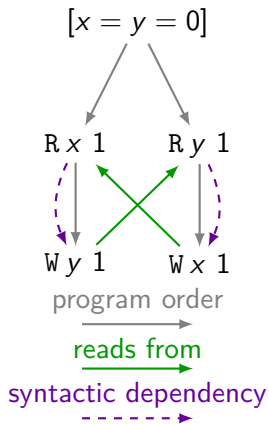
Keep track of **syntactic dependencies** and forbid **dependency cycles**.

Load-buffering

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := 1;$		$x := b;$

Load-buffering + data dependency

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := a;$		$x := b;$



The hardware solution

Keep track of **syntactic dependencies** and forbid **dependency cycles**.

Load-buffering

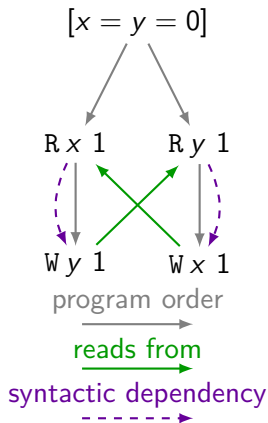
$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := 1;$		$x := b;$

Load-buffering + data dependency

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := a;$		$x := b;$

Load-buffering + fake dependency

$a := x; \text{ // } 1$	\parallel	$b := y; \text{ // } 1$
$y := a + 1 - a;$		$x := b;$



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

The “out-of-thin-air” problem

- ▶ The C/C++11 model is **too weak**:
 - ▶ Values might appear *out-of-thin-air*.
 - ▶ The *DRF guarantee* is broken.
- ▶ A straightforward solution:
 - ▶ Disallow `po` `U` `rf` cycles
 - ▶ But, on weak hardware it carries a certain *implementation cost*.
- ▶ Solving the problem without changing the compilation schemes will require a **major revision** of the standard.

A ‘promising’ solution to OOTA

[Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, Derek Dreyer POPL'17]

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

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

$x = y = 0$	
$x = 1;$	$y = 1;$
$a = y; \text{ // } 0$	$b = x; \text{ // } 0$

Simple operational semantics for C11's relaxed accesses

Store-buffering

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

Memory

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

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

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

T_1 's view

x	y
0	0
1	

T_2 's view

x	y
0	0

- ▶ 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

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

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

$\langle location : value @ timestamp \rangle$

- 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

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

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

$\langle location : value @ timestamp \rangle$

- ▶ 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

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

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

$\langle location : value @ timestamp \rangle$

- ▶ 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

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

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

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

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

⟨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
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

⟨x : 0@0⟩
⟨x : 1@1⟩

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

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

T₁'s view

x	y
0	0
1	

T₂'s view

x	y
0	0
	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⟩

T₁'s view

x
0
1

T₂'s view

x
0
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
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

⟨x : 0@0⟩
⟨x : 1@1⟩
⟨x : 2@2⟩

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

Memory

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

T₁'s view

x	y
0	0
1	

T₂'s view

x	y
0	0
	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⟩

T₁'s view

x
0
1
2

T₂'s view

x
0
2

Load-buffering

$a := x; \text{ // } 1$	\parallel	$x := y;$
$y := 1;$		

- ▶ 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

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

- 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

```
► 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

```
► 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
	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

```
► 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

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

```
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
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

```
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
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

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

Memory

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

T_1 's view

x	y
0	0
1	1

T_2 's view

x	y
0	0
1	1

Load-buffering + dependency

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

Must not admit the same execution!

Promises

Load-buffering

```

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)

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

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

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

Load buffering + fake dependency

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

Load buffering + dependency

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

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

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.

Is this behavior possible?

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

Is this behavior possible?

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

Yes. And the ARM model allows it!

Is this behavior possible?

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

Yes. And the ARM model allows it!

This behavior can be also explained by sequentialization:

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

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)

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)



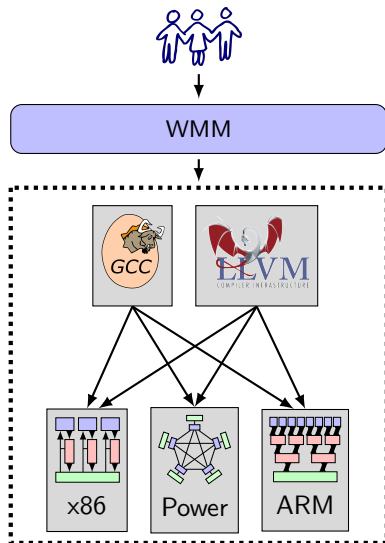
The **Coq**
proof assistant



Results

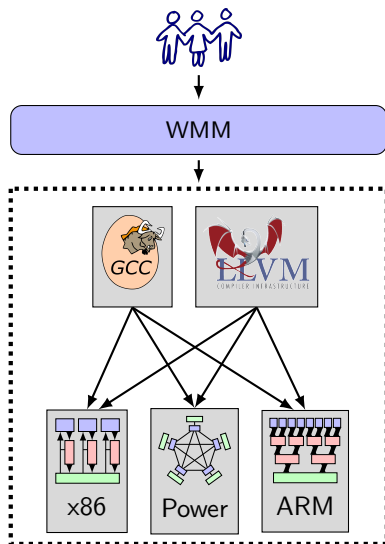
- ▶ No “out-of-thin-air” values
- ▶ DRF guarantees
- ▶ Efficient h/w mappings (x86-TSO, Power, ARM)
- ▶ Compiler optimizations (incl. reorderings, eliminations)

Summary



- ▶ The challenges in designing a WMM.
- ▶ The C/C++11 model.
- ▶ C/C++11 is **broken**:
 - ▶ Most problems are **locally fixable**.
 - ▶ But ruling out **OOTA** requires an entirely different approach.
- ▶ The **promising model** may be the solution.

Summary



- ▶ The challenges in designing a WMM.
- ▶ The C/C++11 model.
- ▶ C/C++11 is **broken**:
 - ▶ Most problems are **locally fixable**.
 - ▶ But ruling out **OOTA** requires an entirely different approach.
- ▶ The **promising model** may be the solution.

Thank you!

<http://www.cs.tau.ac.il/~orilahav/>