

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?

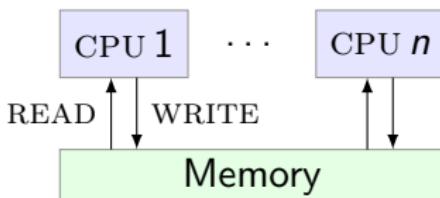
Example: Dekker's mutual exclusion

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):



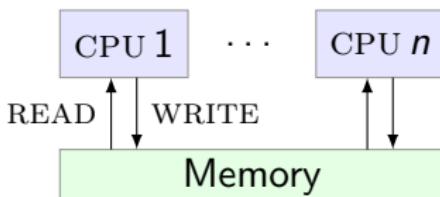
Example: Dekker's mutual exclusion

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;  
}
```

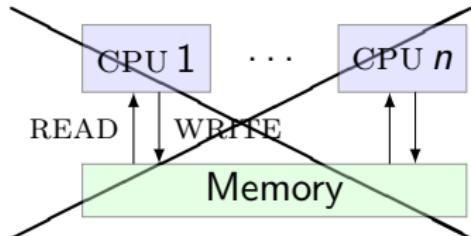
Example: Shared-memory concurrency in C++

```
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?

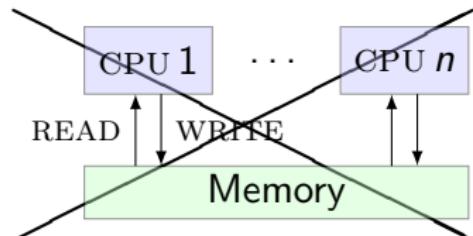
Typically called a **weak memory model (WMM)**

- ▶ 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

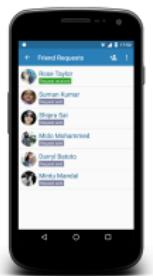
Plan for rest of the talk

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

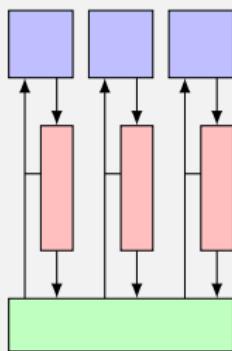
Plan for rest of the talk

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

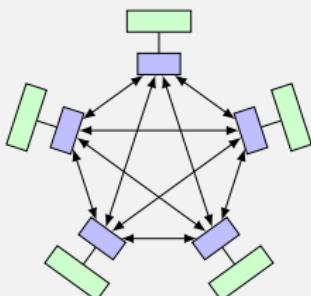
Challenge 1: Various hardware models



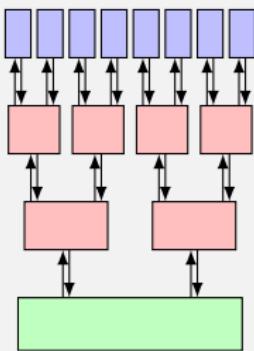
x86-TSO (2010)



POWER (2011)



ARMv8 (2016)



Store buffering in x86-TSO



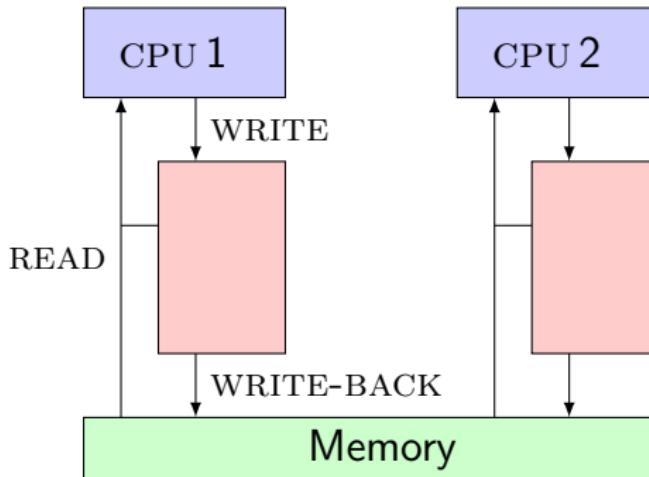
Initially, $x = y = 0$.

$x := 1;$

$y := 1;$

$a := y; // 0$

$b := x; // 0$



Store buffering in x86-TSO



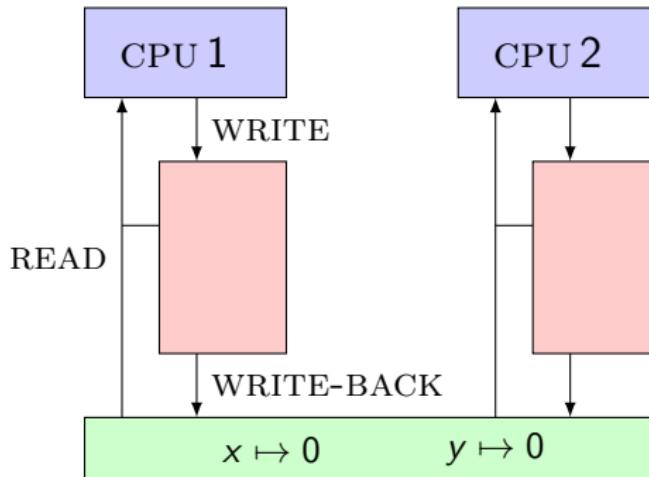
Initially, $x = y = 0$.

► $x := 1;$

$a := y; // 0$

► $y := 1;$

$b := x; // 0$

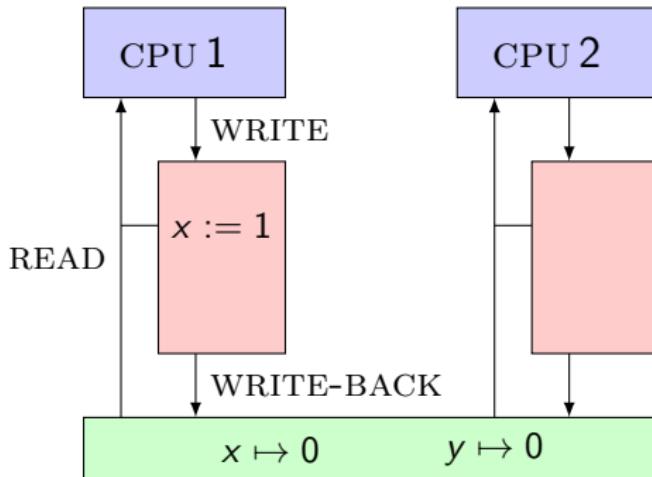


Store buffering in x86-TSO



Initially, $x = y = 0$.

```
x := 1;           ► y := 1;  
► a := y; // 0    b := x; // 0
```

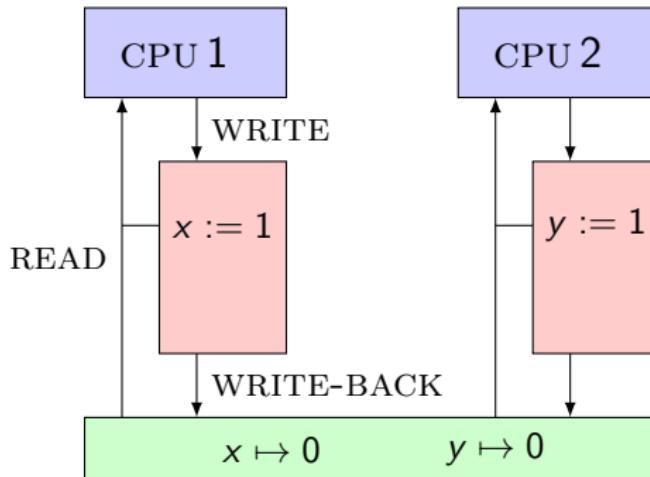


Store buffering in x86-TSO



Initially, $x = y = 0$.

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



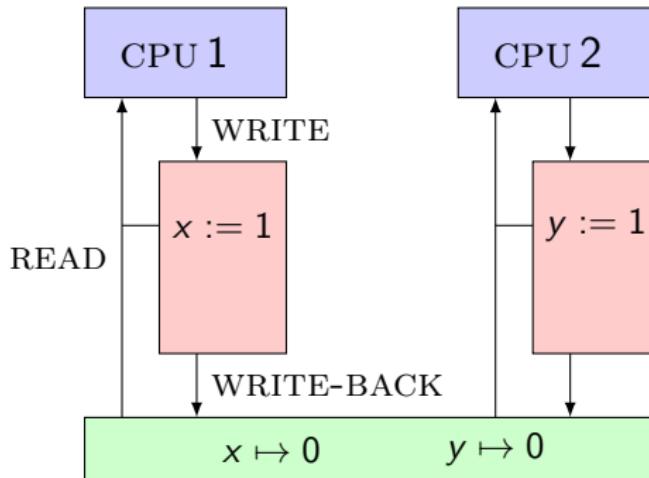
Store buffering in x86-TSO



Initially, $x = y = 0$.

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

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

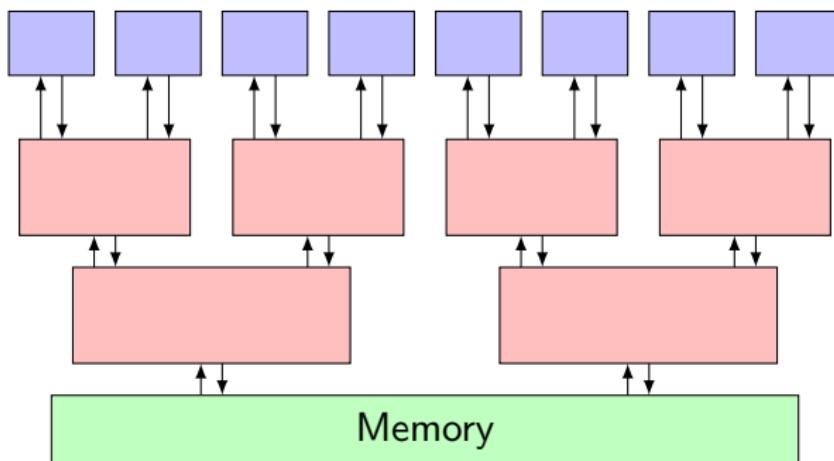


Load buffering in ARM



Initially, $x = y = 0$.

$a := x; \# 1$		$b := y; \# 1$
$y := 1;$		$x := b;$

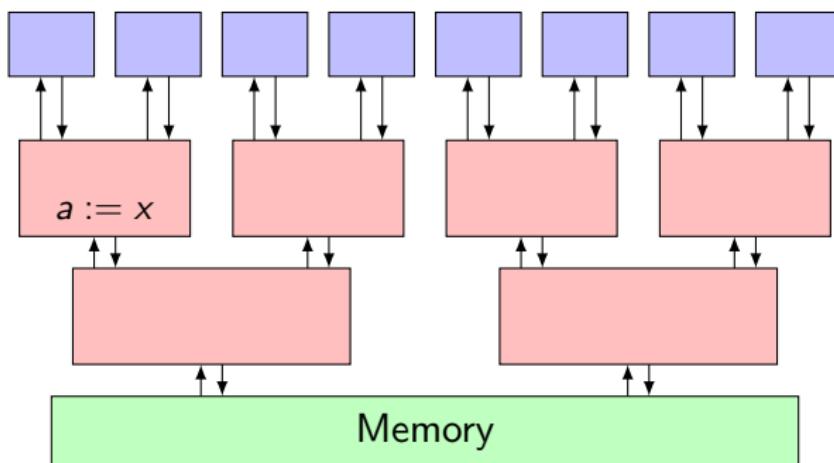


Load buffering in ARM



Initially, $x = y = 0$.

$a := x; \# 1$ || $b := y; \# 1$
 $y := 1;$ $x := b;$

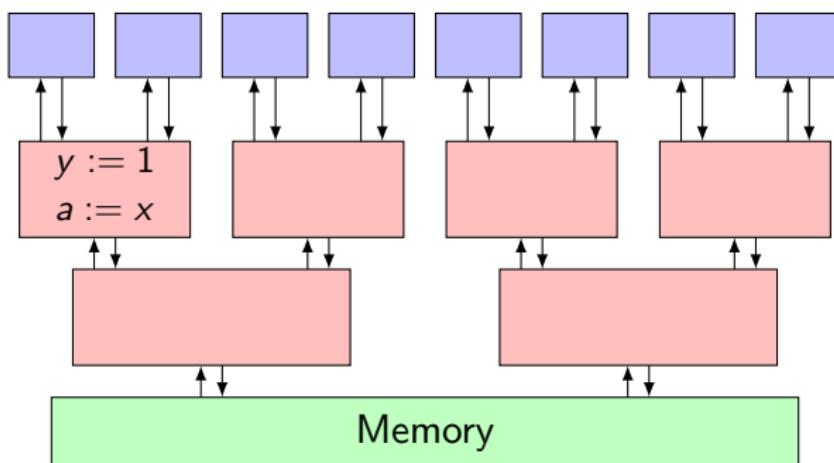


Load buffering in ARM



Initially, $x = y = 0$.

$a := x; \# 1$ || $b := y; \# 1$
 $y := 1;$ $x := b;$

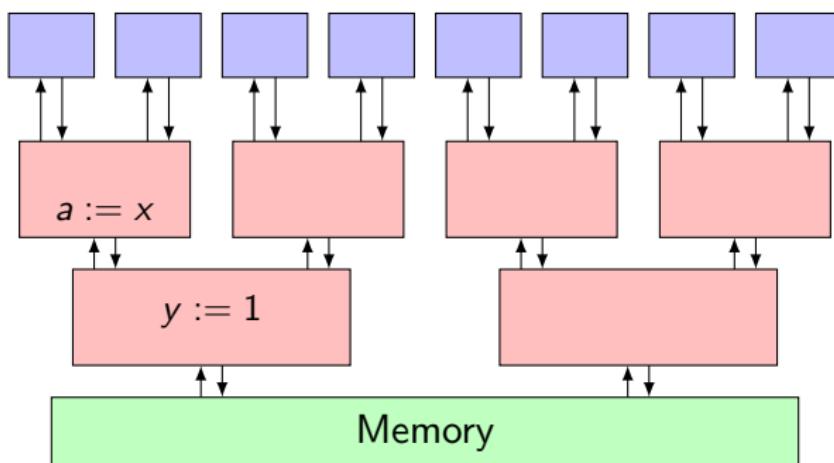


Load buffering in ARM



Initially, $x = y = 0$.

$a := x; \# 1$ || $b := y; \# 1$
 $y := 1;$ $x := b;$

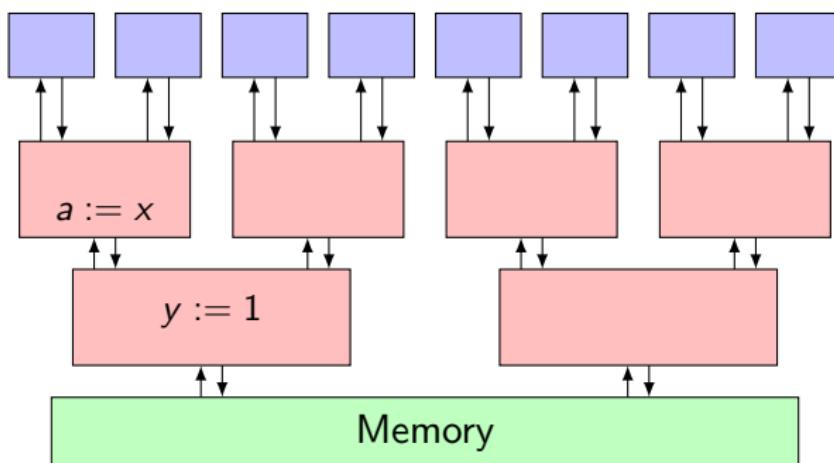


Load buffering in ARM



Initially, $x = y = 0$.

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



Challenge 2: Compilers stir the pot

Initially, $x = y = 0$.

$x := 1;$	$a := x;$
$y := 1;$	$b := y; // 1$
	$c := x; // 0$

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)

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

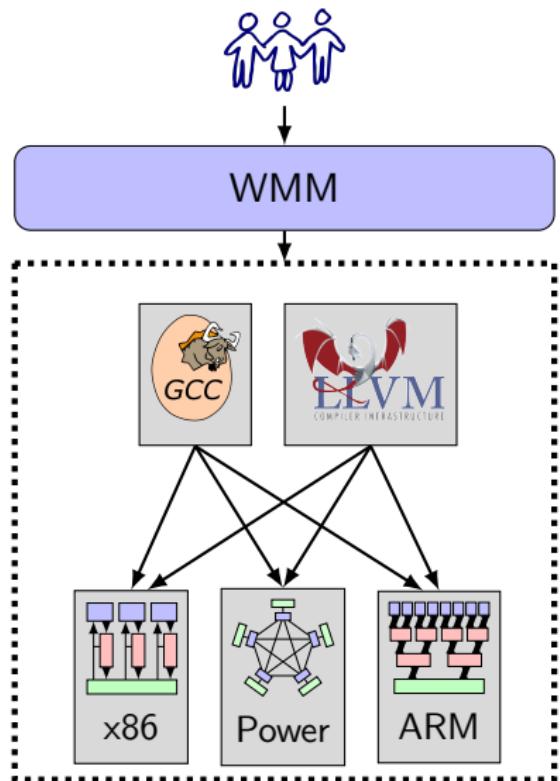
Independent reads of independent writes (IRIW)

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

ARM-weak

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

Overview



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

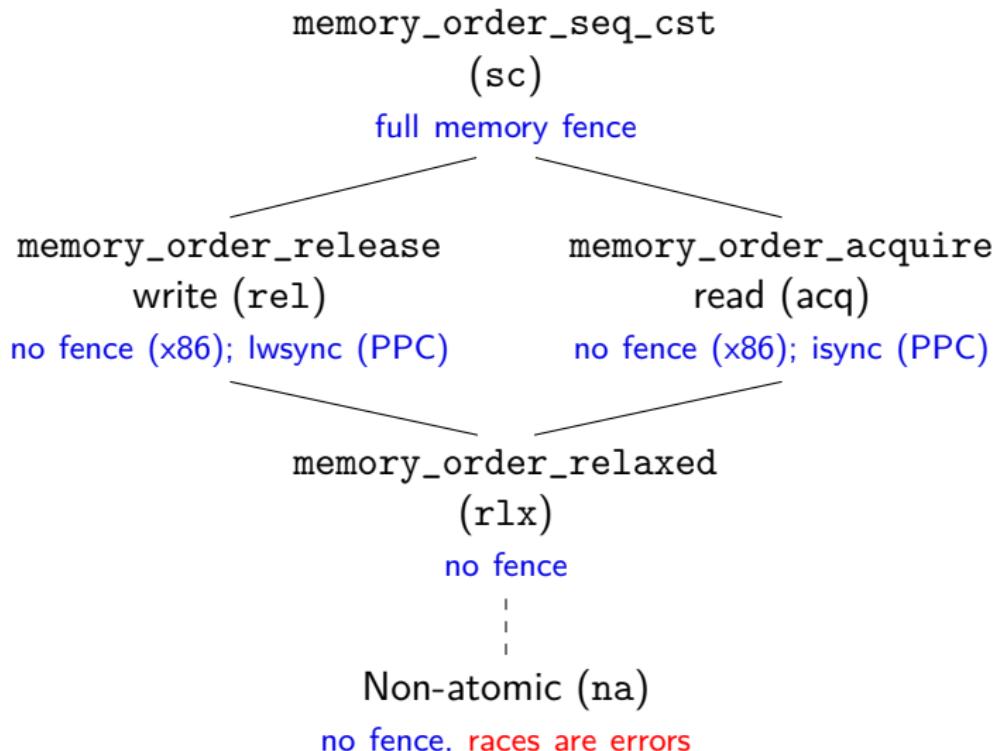
Two types of accesses



DRF (data race freedom) guarantee

no data races under SC \implies only SC behaviors

A spectrum of access modes



+ Explicit primitives for fences

C11: a declarative memory model

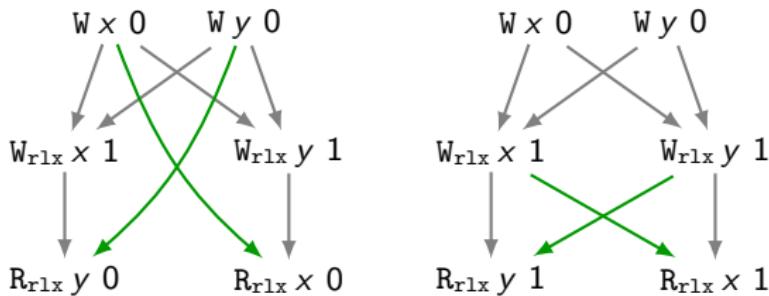
Declarative semantics abstracts away from implementation details.

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)

$$\begin{array}{c} x = y = 0 \\ x :=_{\text{rlx}} 1 \parallel y :=_{\text{rlx}} 1 \\ a := y_{\text{rlx}} \quad b := x_{\text{rlx}} \end{array}$$



Relations

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

C/C++11 formal model

[Vafeiadis & Narayan OOPSLA'13]

$\llbracket - \rrbracket : \text{CExp} \rightarrow \mathbb{P}((\text{res} : \text{Val} \cup \{\perp\}, \mathcal{A} : \mathbb{P}(\text{AName}), \text{lab} : \mathcal{A} \rightarrow \text{Act}, \text{sb} : \mathbb{P}(\mathcal{A} \times \mathcal{A}), \text{fst} : \mathcal{A}, \text{lst} : \mathcal{A}))$
$\llbracket v \rrbracket \stackrel{\text{def}}{=} \{\langle v, \{a\}, \text{lab}, \emptyset, a, a \rangle \mid a \in \text{AName} \wedge \text{lab}(a) = \text{skip}\}$
$\llbracket \text{alloc}() \rrbracket \stackrel{\text{def}}{=} \{\langle \ell, \{a\}, \text{lab}, \emptyset, a, a \rangle \mid a \in \text{AName} \wedge \ell \in \text{Loc} \wedge \text{lab}(a) = A(\ell)\}$
$\llbracket [v]_Z := v' \rrbracket \stackrel{\text{def}}{=} \{\langle v', \{a\}, \text{lab}, \emptyset, a, a \rangle \mid a \in \text{AName} \wedge \text{lab}(a) = W_Z(v, v')\}$
$\llbracket [v]_Z \rrbracket \stackrel{\text{def}}{=} \{\langle v', \{a\}, \text{lab}, \emptyset, a, a \rangle \mid a \in \text{AName} \wedge v' \in \text{Val} \wedge \text{lab}(a) = R_Z(v, v')\}$
$\llbracket \text{CAS}_{X,Y}(v, v_o, v_n) \rrbracket \stackrel{\text{def}}{=} \{\langle v', \{a\}, \text{lab}, \emptyset, a, a \rangle \mid a \in \text{AName} \wedge v' \in \text{Val} \wedge v' \neq v_o \wedge \text{lab}(a) = \text{R}_Y(v, v')\}$
$\cup \{\langle v_o, \{a\}, \text{lab}, \emptyset, a, a \rangle \mid a \in \text{AName} \wedge \text{lab}(a) = \text{RMW}_{X,Y}(v, v_o, v_n)\}$
$\llbracket \text{let } x = E_1 \text{ in } E_2 \rrbracket \stackrel{\text{def}}{=} \{\langle \perp, A_1, \text{lab}_1, s_{b_1}, f_{st_1}, l_{st_1} \rangle \mid \langle \perp, A_1, \text{lab}_1, s_{b_1}, f_{st_1}, l_{st_1} \rangle \in [E_1]\}$
$\cup \{\langle res_2, A_1 \uplus A_2, \text{lab}_1 \uplus \text{lab}_2, s_{b_1} \uplus s_{b_2} \cup \{(l_{st_1}, f_{st_2})\}, f_{st_1}, l_{st_2} \mid$
$\langle v_1, A_1, \text{lab}_1, s_{b_1}, f_{st_1}, l_{st_1} \rangle \in [E_1] \wedge \langle res_2, A_2, \text{lab}_2, s_{b_2}, f_{st_2}, l_{st_2} \rangle \in [E_2[v_1/x]]\}$
$\llbracket \text{repeat } E \text{ end} \rrbracket \stackrel{\text{def}}{=} \{\langle res_N, \bigcup_{i \in [1..N]} A_i, \bigcup_{i \in [1..N]} \text{lab}_i, \bigcup_{i \in [1..N]} s_{b_i} \cup \{(l_{st_1}, f_{st_2}), \dots, (l_{st_{N-1}}, f_{st_N})\}, f_{st_1}, l_{st_N} \mid$
$\forall i, \langle res_i, A_i, \text{lab}_i, s_{b_i}, f_{st_i}, l_{st_i} \rangle \in [E] \wedge (i \neq N \implies res_i = 0) \wedge res_N \neq 0\}$
$\llbracket E_1 \parallel E_2 \rrbracket \stackrel{\text{def}}{=} \{\langle \text{combine}(\langle res_1, res_2 \rangle, A_1 \uplus A_2 \uplus \{a_{\text{fork}}, a_{\text{join}}\}), \text{lab}_1 \uplus \text{lab}_2 \cup \{a_{\text{fork}} \mapsto \text{skip}, a_{\text{join}} \mapsto \text{skip}\},$
$s_{b_1} \uplus s_{b_2} \cup \{a_{\text{fork}}, f_{st_1}, (a_{\text{fork}}, f_{st_2}), (l_{st_1}, a_{\text{join}}), (l_{st_2}, a_{\text{join}})\}, a_{\text{fork}}, a_{\text{join}} \mid$
$\langle res_1, A_1, s_{b_1}, f_{st_1}, l_{st_1} \rangle \in [E_1] \wedge \langle res_2, A_2, s_{b_2}, f_{st_2}, l_{st_2} \rangle \in [E_2] \wedge a_{\text{fork}}, a_{\text{join}} \in \text{AName}\}$

Figure 2. Semantics of closed program expressions.

$\exists x. \text{hb}(x, x)$	(IrreflexiveHB)
$\forall \ell. \text{totalorder}(\{a \in \mathcal{A} \mid \text{iswrite}_\ell(a)\}, \text{mo}) \wedge \text{hb}_\ell \subseteq \text{mo}$	(ConsistentMO)
$\text{totalorder}(\{a \in \mathcal{A} \mid \text{isSeqCst}(a)\}, \text{sc}) \wedge \text{hb}_{\text{SeqCst}} \subseteq \text{sc} \wedge \text{mo}_{\text{SeqCst}} \subseteq \text{sc}$	(ConsistentSC)
$\forall b. \text{rf}(b) \neq \perp \iff \exists \ell. a. \text{iswrite}_\ell(a) \wedge \text{isread}_\ell(b) \wedge \text{hb}(a, b)$	(ConsistentRFdom)
$\forall a, b. \text{rf}(b) = a \implies \exists \ell. v. \text{iswrite}_{\ell,v}(a) \wedge \text{isread}_{\ell,v}(b) \wedge \neg \text{hb}(b, a)$	(ConsistentRF)
$\forall a, b. \text{rf}(b) = a \wedge (\text{mode}(a) = \text{na} \vee \text{mode}(b) = \text{na}) \implies \text{hb}(a, b)$	(ConsistentRFna)
$\forall a, b. \text{rf}(b) = a \wedge \text{isSeqCst}(b) \implies \text{isc}(a, b) \vee \neg \text{isSeqCst}(a) \wedge (\forall x. \text{isc}(x, b) \Rightarrow \neg \text{hb}(a, x))$	(RestrSCReads)
$\nexists a, b. \text{hb}(a, b) \wedge \text{mo}(\text{rf}(b), \text{rf}(a)) \wedge \text{locs}(a) = \text{locs}(b)$	(CoherentRR)
$\nexists a, b. \text{hb}(a, b) \wedge \text{mo}(\text{rf}(b), a) \wedge \text{iswrite}(a) \wedge \text{locs}(a) = \text{locs}(b)$	(CoherentWR)
$\nexists a, b. \text{hb}(a, b) \wedge \text{mo}(a, \text{rf}(b)) \wedge \text{iswrite}(b) \wedge \text{locs}(a) = \text{locs}(b)$	(CoherentRW)
$\forall a. \text{isrmw}(a) \wedge \text{rf}(a) \neq \perp \implies \text{mo}(\text{rf}(a), a) \wedge \nexists c. \text{mo}(\text{rf}(a), c) \wedge \text{mo}(c, a)$	(AtomicRMW)
$\forall a, b, \ell. \text{lab}(a) = \text{lab}(b) = A(\ell) \implies a = b$	(ConsistentAlloc)
where $\text{iswrite}_{\ell,v}(a) \stackrel{\text{def}}{=} \exists X, v_{\text{old}}. \text{lab}(a) \in \{\text{WX}(\ell, v), \text{RMW}(\ell, v_{\text{old}}, v)\}$	
$\text{iswrite}_\ell(a) \stackrel{\text{def}}{=} \exists X, v_{\text{new}}. \text{lab}(a) \in \{\text{RX}(\ell, v), \text{RMW}(\ell, v, v_{\text{new}})\}$	etc.
$\text{rsElem}(a, b) \stackrel{\text{def}}{=} \text{sameThread}(a, b) \vee \text{isrmw}(b)$	
$\text{rseq}(a) \stackrel{\text{def}}{=} \{a\} \cup \{b \mid \text{rsElem}(a, b) \wedge \text{mo}(a, b) \wedge (\forall c. \text{mo}(a, c) \wedge \text{mo}(c, b) \Rightarrow \text{rsElem}(a, c))\}$	
$\text{sw} \stackrel{\text{def}}{=} \{(a, b) \mid \text{mode}(a) \in \{\text{rel}, \text{rel_acq}, \text{sc}\} \wedge \text{mode}(b) \in \{\text{acq}, \text{rel_acq}, \text{sc}\} \wedge \text{rf}(b) \in \text{rseq}(a)\}$	
$\text{hb} \stackrel{\text{def}}{=} (\text{sb} \cup \text{sw})^+$	
$\text{hb}_\ell \stackrel{\text{def}}{=} \{(a, b) \in \text{hb} \mid \text{iswrite}_\ell(a) \wedge \text{iswrite}_\ell(b)\}$	
$X_{\text{SeqCst}} \stackrel{\text{def}}{=} \{(a, b) \in X \mid \text{isSeqCst}(a) \wedge \text{isSeqCst}(b)\}$	
$\text{isc}(a, b) \stackrel{\text{def}}{=} \text{iswrite}_{\text{locx}(b)}(a) \wedge \text{sc}(a, b) \wedge \nexists c. \text{sc}(a, c) \wedge \text{sc}(c, b) \wedge \text{iswrite}_{\text{locx}(b)}(c)$	

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

$c : W(\ell, 1) \xrightarrow[\text{mo}]{\text{rf}} a : R(\ell, 1)$	$c : W(\ell, 2) \xrightarrow[\text{mo}]{\text{rf}} a : W(\ell, 1)$	$c : W(\ell, 1) \xrightarrow[\text{mo}]{\text{rf}} a : R(\ell, 1)$	$a \xrightarrow{\text{rf}} b \text{ means } a = \text{rf}(b)$
\uparrow_{mo}	\downarrow_{rf}	\uparrow_{mo}	\downarrow_{mo}
$d : W(\ell, 2) \xrightarrow[\text{rf}]{\text{rb}} b : R(\ell, 2)$	$b : R(\ell, 2)$	$b : W(\ell, 2)$	$a \xrightarrow{\text{mo}} b \text{ means } \text{mo}(a, b)$
violates CoherentRR	violates CoherentWR	violates CoherentRW	$a \xrightarrow{\text{hb}} b \text{ means } \text{hb}(a, b)$

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 $\text{po} \cup \text{rf} \subseteq \text{sc}$.
- ▶ If $\langle a, b \rangle \in \text{rf}$ then there does not exist $c \in W_{\text{loc}(a)}$ such that $\langle a, c \rangle \in \text{sc}$ and $\langle c, b \rangle \in \text{sc}$.

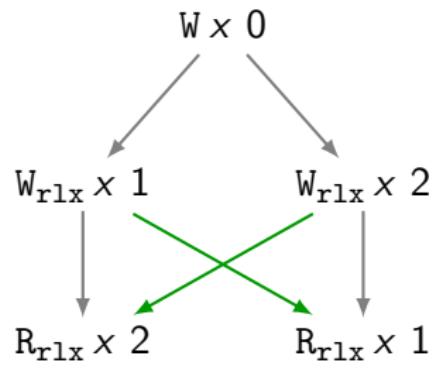
Definition (SC-per-location)

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 $\text{po} \cup \text{rf} \subseteq \text{sc}_x$.
- ▶ If $\langle a, b \rangle \in \text{rf}$ then there does not exist $c \in W_x$ such that $\langle a, c \rangle \in \text{sc}_x$ and $\langle c, b \rangle \in \text{sc}_x$.

SC-per-location: Example

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



program order
reads from

inconsistent!

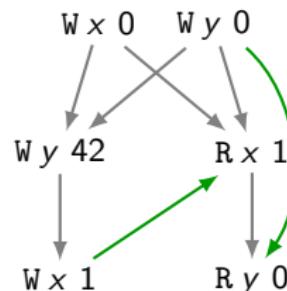
Release/acquire synchronization

SC-per-location is often too weak:

- ▶ It does not support the message passing idiom:

Message passing (MP)

```
y := 42;    // 1  
x := 1;    // 0
```



- ▶ We cannot even implement locks:

Simple lock

```
lock();  
x := 1;  
x := 2;  
unlock();
```

lock();
a := x; // 1
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;    || race   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);  
        }||
```

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
```

4

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

3

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

4

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

3

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

4

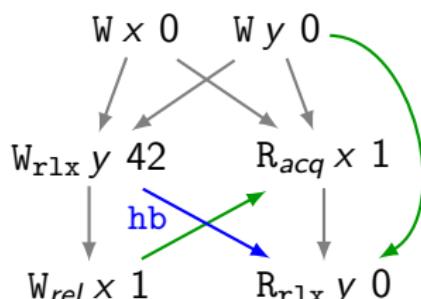
```
int y = 0;  
atomic<int> x = 0;  
y = 42; || if(xrlx == 1){  
fencerel; || rf      fenceacq  
x =rlx 1; || sw      print(y);  
        }||
```

The “happens-before” relation

Definition (happens-before)

$$\begin{array}{c} \text{a: W Rel} \qquad \text{b: R Acq} \\ \hline \begin{array}{c} a \xrightarrow{\text{po}} b \\ \hline a \xrightarrow{\text{hb}} b \end{array} \qquad \begin{array}{c} a \xrightarrow{\text{rf}} b \\ \hline a \xrightarrow{\text{hb}} b \end{array} \qquad \begin{array}{c} a \xrightarrow{\text{hb}} b \qquad b \xrightarrow{\text{hb}} c \\ \hline a \xrightarrow{\text{hb}} c \end{array} \end{array}$$

- ▶ **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

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

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

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

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!)

```
a := xacq; // 1 || x :=sc 1; || y :=sc 1; || c := yacq; // 1  
b := ysc; // 0
```

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

```
a := xacq; // 1 || x :=rel 1; || y :=rel 1; || c := yacq; // 1  
fence-sc; || d := xacq; // 0
```

- ▶ 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 \sqsubset relaxed \sqsubset release/acquire \sqsubset sc

Load-buffering

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

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

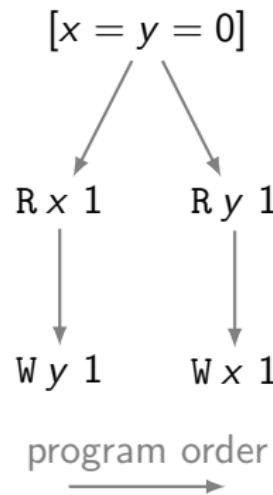
C/C++11 is too weak

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

Load-buffering

$$\begin{array}{c|c} a := x; \text{ // 1} & b := y; \text{ // 1} \\ y := 1; & x := b; \end{array}$$

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



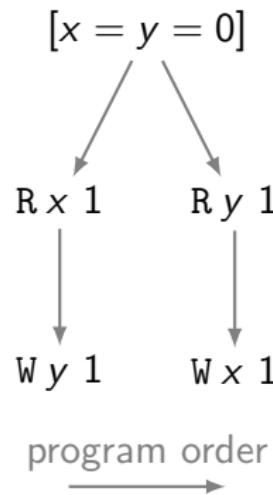
C/C++11 is too weak

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

Load-buffering

$$\begin{array}{c|c} a := x; \text{ // 1} & b := y; \text{ // 1} \\ y := 1; & x := b; \end{array}$$

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



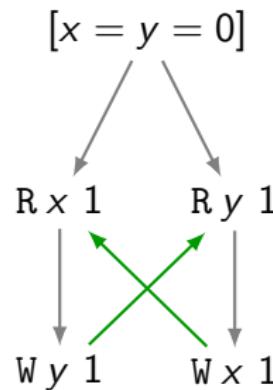
C/C++11 is too weak

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

Load-buffering

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

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



program order
→
reads from
→

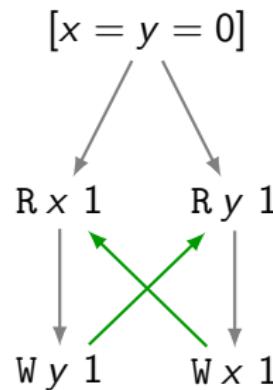
C/C++11 is too weak

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

Load-buffering

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

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



program order
→
reads from
→

C/C++11 is too weak

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

Load-buffering

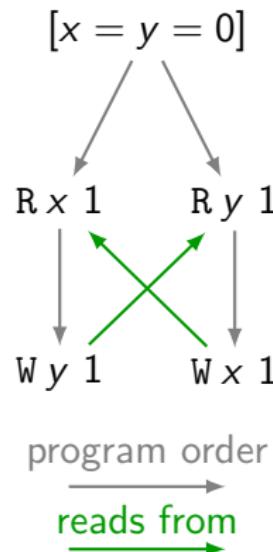
```
a := x; // 1      || b := y; // 1  
y := 1;          || x := b;
```

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

Load-buffering + data dependency

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

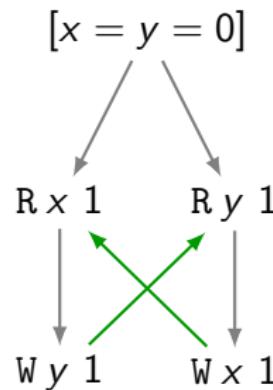
non-atomic \sqsubset relaxed \sqsubset release/acquire \sqsubset sc

Load-buffering + control dependency

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

C/C++11 allows this behavior.

The DRF guarantee is broken!



program order
→
reads from
→

C/C++11 is too weak

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

Load-buffering + control dependency

$a := x; // 1$
if ($a = 1$)

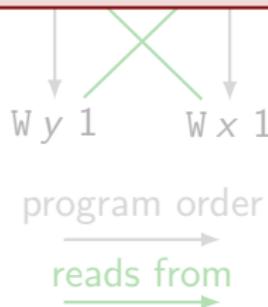
$b := y; // 1$
if ($b = 1$)

$[x = y = 0]$
/ \

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

C/

The DRF guarantee is broken!



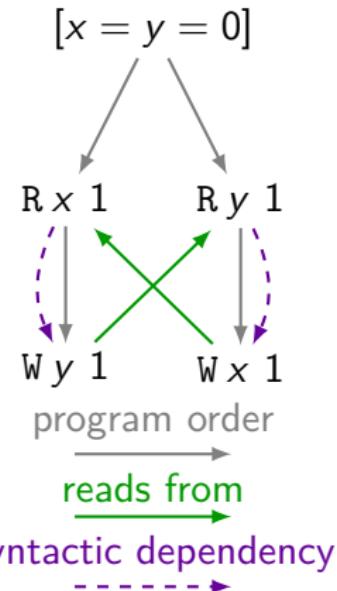
The hardware solution

Keep track of syntactic dependencies and forbid dependency cycles.

Load-buffering

$$\begin{array}{c|c} a := x; \text{ // } 1 & b := y; \text{ // } 1 \\ y := 1; & x := b; \end{array}$$

Load-buffering + data dependency

$$\begin{array}{c|c} a := x; \text{ // } 1 & b := y; \text{ // } 1 \\ y := a; & x := b; \end{array}$$


The hardware solution

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

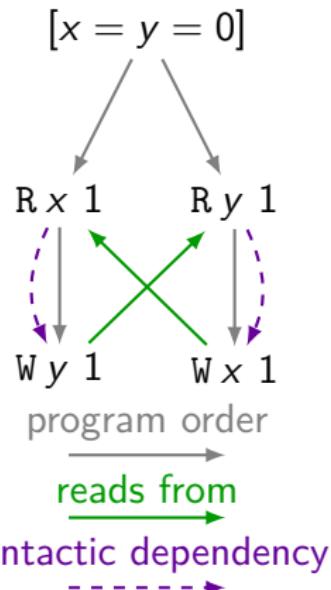
Load-buffering

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

Load-buffering + data dependency

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

Load-buffering + fake dependency

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


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` \cup `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

$$\begin{array}{ccc} x = y = 0 & & \\ \begin{array}{c} x = 1; \\ a = y; \textcolor{green}{//} 0 \end{array} & \parallel & \begin{array}{c} y = 1; \\ b = x; \textcolor{green}{//} 0 \end{array} \end{array}$$

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

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
 $\langle \text{location} : \text{value} @ \text{timestamp} \rangle$
- 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

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

Memory

$$\begin{array}{l} \langle x : 0 @ 0 \rangle \\ \langle y : 0 @ 0 \rangle \\ \langle x : 1 @ 1 \rangle \end{array}$$

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

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

- 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

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

- 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

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


Memory

$$\begin{array}{l} \langle x : 0 @ 0 \rangle \\ \langle y : 0 @ 0 \rangle \\ \langle x : 1 @ 1 \rangle \\ \langle y : 1 @ 1 \rangle \end{array}$$

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

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

- 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

$x = y = 0$

$x = 1; \quad \parallel \quad y = 1;$

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

► ►

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

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

- 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

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



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

$x = 0$
 $x := 1; \quad \parallel \quad x := 2;$
 $a = x; \text{ //2} \quad \parallel \quad b = x; \text{ //1}$

Simple operational semantics for C11's relaxed accesses

Store-buffering

$x = y = 0$

$$\begin{array}{c|c} x = 1; & y = 1; \\ \hline a = y; \textcolor{green}{// 0} & b = x; \textcolor{green}{// 0} \end{array}$$

► ►

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

$x = 0$

$$\begin{array}{c|c} \blacktriangleright x := 1; & \blacktriangleright x := 2; \\ \hline a = x; \textcolor{red}{// 2} & b = x; \textcolor{red}{// 1} \end{array}$$

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

$x = y = 0$

$x = 1; \parallel y = 1;$

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

► ►

Memory

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

T_1 's view

x	y
0	0
	1

T_2 's view

x	y
0	0
	1

Coherence Test

$x = 0$

$x := 1; \parallel \blacktriangleright x := 2;$

► $a = x; // 2 \parallel b = x; // 1$

► ||

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

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

► ►

Memory

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

T_1 's view

x	y
0	0
	1

T_2 's view

x	y
0	0
	1

Coherence Test

$x = 0$
 $x := 1; \parallel x := 2;$
► $a = x; // 2 \parallel$ ► $b = x; // 1$

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

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

► ►

Memory

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

T_1 's view

x	y
0	0
	1

T_2 's view

x	y
0	0
	1

Coherence Test

$x = 0$
 $x := 1; \parallel x := 2;$
 $a = x; // 2 \parallel b = x; // 1$

► ►

Memory

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

T_1 's view

x
0
1
2

T_2 's view

x
0
2

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

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

T_1 's view

x	y
0	0
1	

T_2 's view

x	y
0	0
	1

Coherence Test

$x = 0$
 $x := 1; \parallel x := 2;$
 $a = x; // 2 \parallel b = x; // 1$



Memory

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

T_1 's view

x
0
1
2

T_2 's view

x
2

Promises

Load-buffering

```
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

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

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
X
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

T_2 's view

x	y
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
✗	✗
1	1

T_2 's view

x	y
✗	✗
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 || 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
1	1

T_2 's view

x	y
1	1

Load-buffering + dependency

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

Must not admit the same execution!

Promises

Load-buffering

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

Key Idea

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

Load-buffering + dependency

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

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

$$a := x; \textcolor{green}{// 1} \quad \| \quad y := 1; \quad \| \quad x := y;$$

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

Load buffering + dependency

$$a := x; \textcolor{red}{// 1} \quad \| \quad y := a; \quad \| \quad x := y;$$

Load buffering + fake dependency

$$a := x; \textcolor{green}{// 1} \quad \| \quad y := a + 1 - a; \quad \| \quad x := y;$$

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

Quiz

Is this behavior possible?

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

Quiz

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

Is this behavior possible?

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

Quiz

Is this behavior possible?

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

Yes. And the ARM model allows it!

Quiz

Is this behavior possible?

$$a := x; \textcolor{green}{\#1} \parallel x := 1; \qquad y := x; \qquad x := y;$$

Yes. And the ARM model allows it!

This behavior can be also explained by sequentialization:

$$a := x; \textcolor{green}{\#1} \parallel x := 1; \qquad y := x; \qquad x := y; \quad \leadsto \quad x := 1; \qquad y := x; \qquad a := x; \textcolor{green}{\#1} \parallel x := y;$$

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)



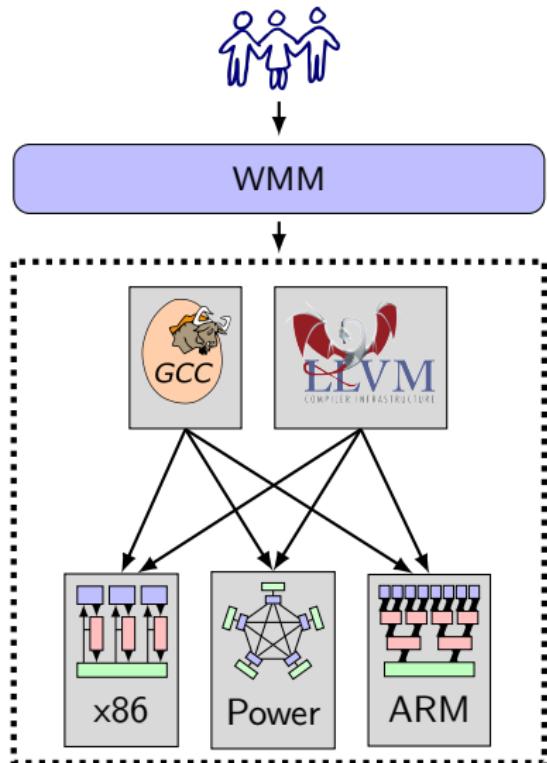
The **Coq**
proof assistant



Results

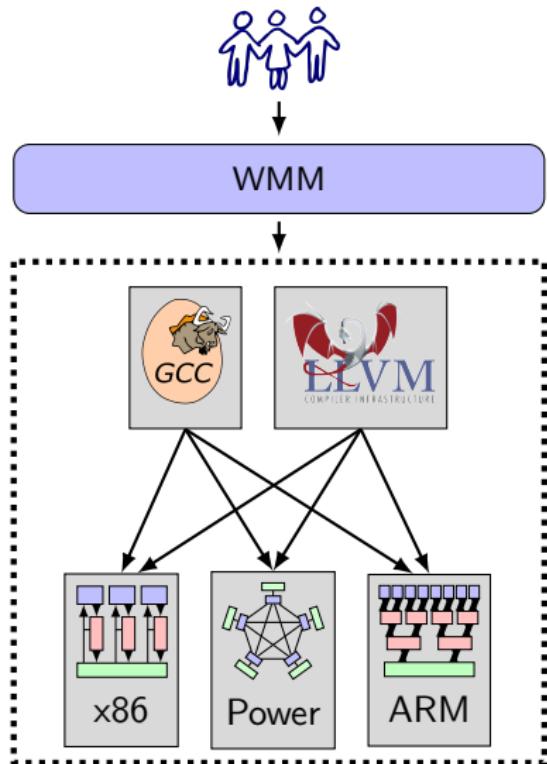
- ▶ No “out-of-thin-air” values
- ▶ DRF guarantees
- ▶ Efficient h/w mappings (x86-TSO, Power, ARM)
- ▶ Compiler optimizations (incl. reorderings, eliminations)

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/>