Weak Memory Concurrency in C/C++11

Ori Lahav

Hydra
Distributed computing conference

July 11, 2019

Example: Dekker's mutual exclusion

Initially, $x = y = 0$.

$x := 1;$	$y := 1;$
$a := y;$	$b := x;$
if $(a = 0)$ **then**	**if** $(b = 0)$ **then**
/* *critical section* */	/* *critical section* */

Example: Dekker's mutual exclusion

Initially, $x = y = 0$.

$x := 1;$	$y := 1;$
$a := y;$	$b := x;$
if $(a = 0)$ **then**	**if** $(b = 0)$ **then**
/* *critical section* */	/* *critical section* */

Is it safe?

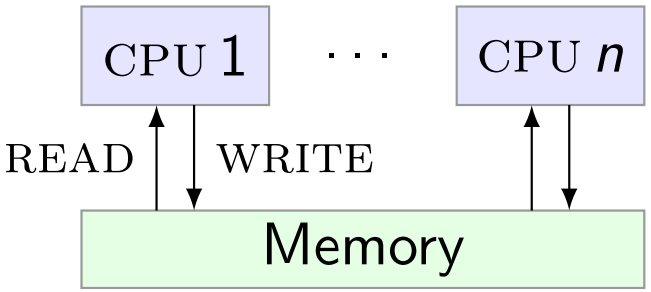
Example: Dekker's mutual exclusion

Initially, $x = y = 0$.

$x := 1;$	$y := 1;$
$a := y;$ // 0	$b := x;$ // 0
if $(a = 0)$ **then**	**if** $(b = 0)$ **then**
/* *critical section* */	/* *critical section* */

Is it safe?

Yes, if we assume sequential consistency (SC):



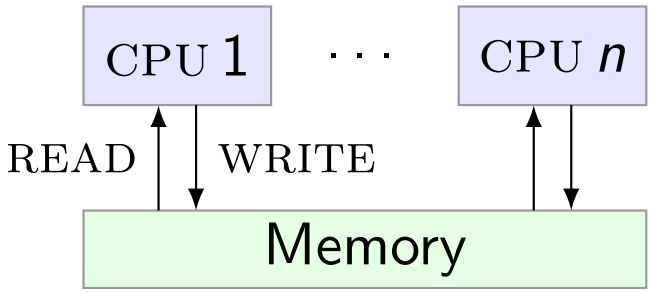
Example: Dekker's mutual exclusion

Initially, $x = y = 0$.

$x := 1;$	$y := 1;$
$a := y;$ // 0	$b := x;$ // 0
if $(a = 0)$ **then**	**if** $(b = 0)$ **then**
/* critical section */	/* 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;
}
```

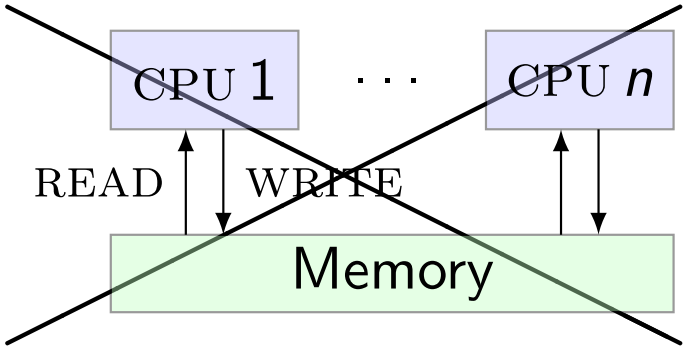
Example: Shared-memory concurrency in C++

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

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

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

```
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;
}
```

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

Weak memory models



We look for a *substitute for SC:*

Unambiguous specification

- What are the possible outcomes of a multithreaded program?

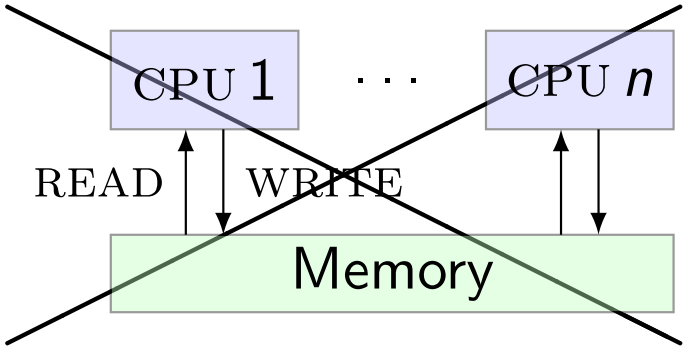
Amenable to formal reasoning

- Can prove theorems about the model.

Typically called a weak memory model (WMM)

- Allows more behaviors than SC.

Weak memory models



We look for a *substitute for SC:*

Unambiguous specification

- What are the possible outcomes of a multithreaded program?

Amenable to formal reasoning

- Can prove theorems about the model.

Typically called a weak memory model (WMM)

- Allows more behaviors than SC.

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

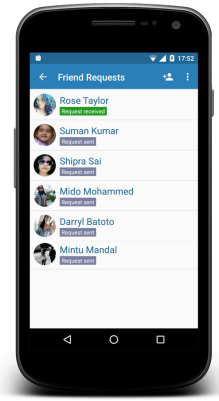
Plan for rest of the talk

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

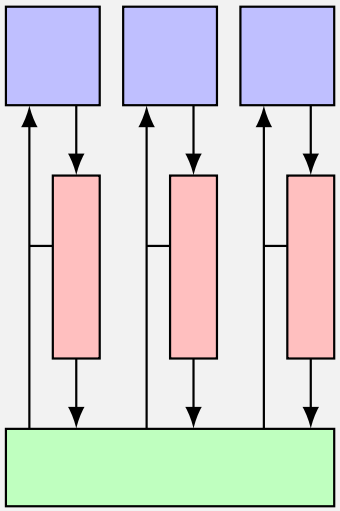
Plan for rest of the talk

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

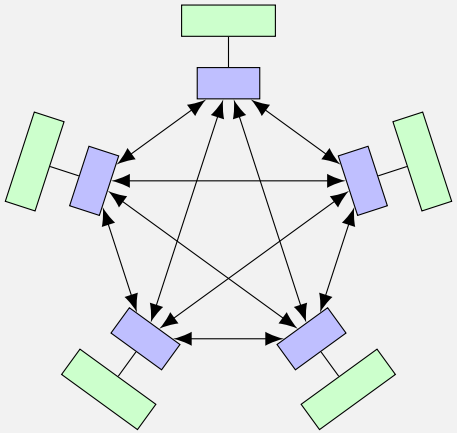
Challenge 1: Various hardware models



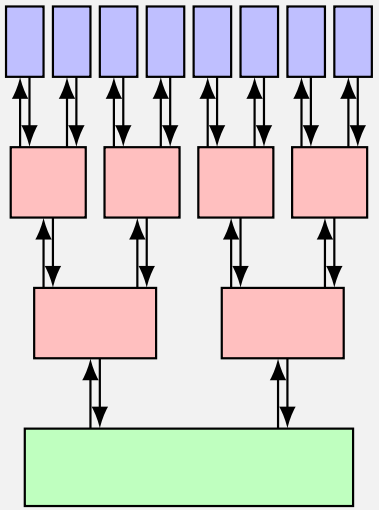
x86-TSO (intel) AMD
(2010)



POWER IBM
(2011)



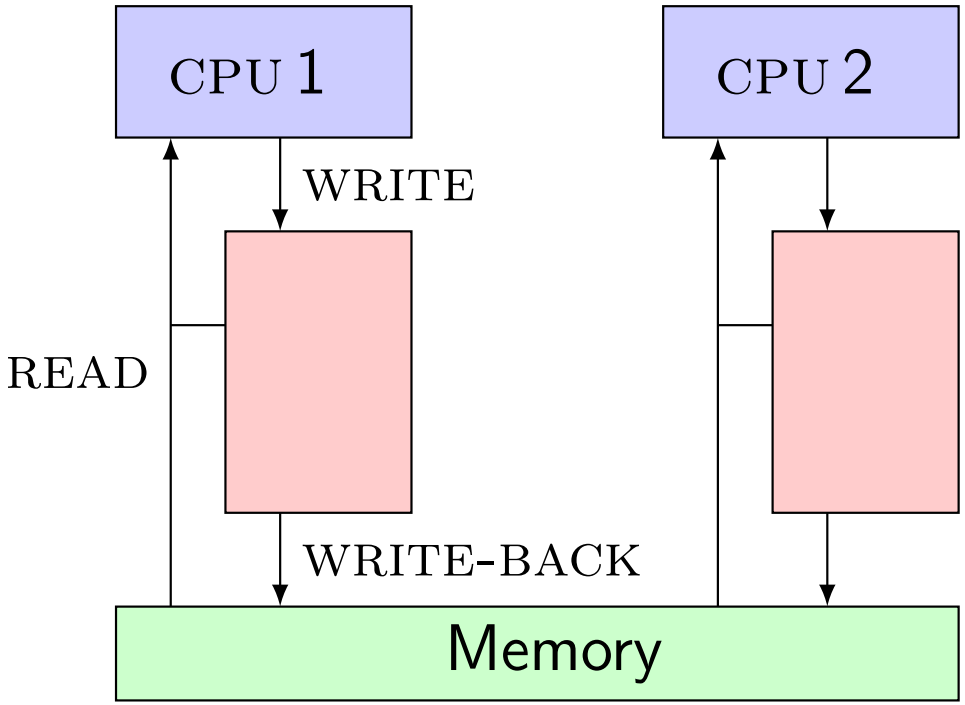
ARMv8 ARM
(2016)



Store buffering in x86-TSO

Initially, $x = y = 0$.

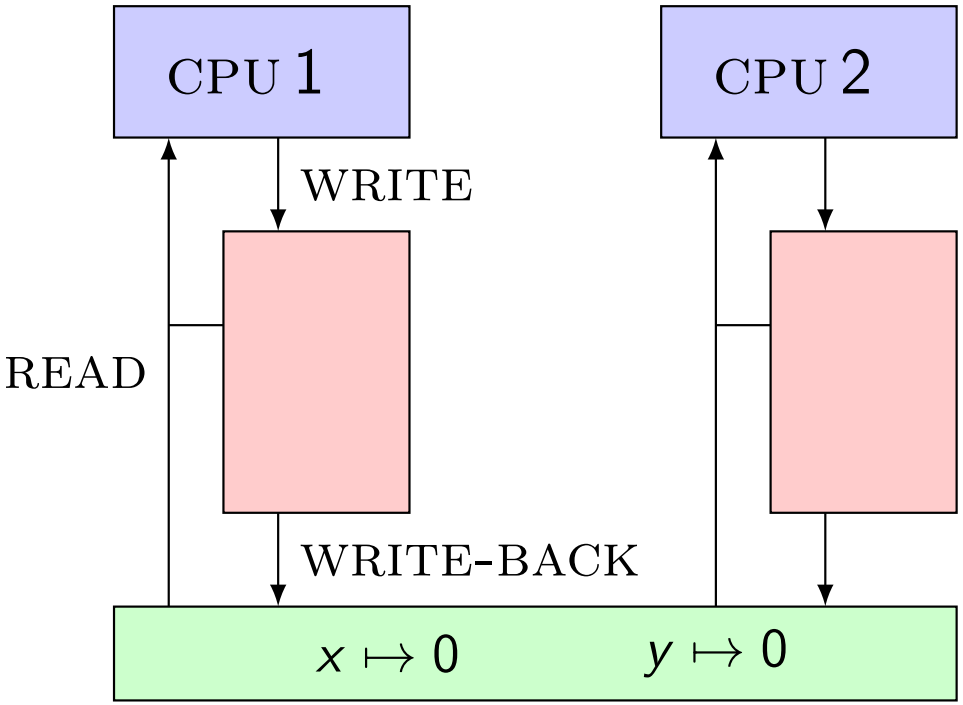
$x := 1;$	$\parallel$	$y := 1;$
$a := y;$ // 0	$\parallel$	$b := x;$ // 0



Store buffering in x86-TSO

Initially, $x = y = 0$.

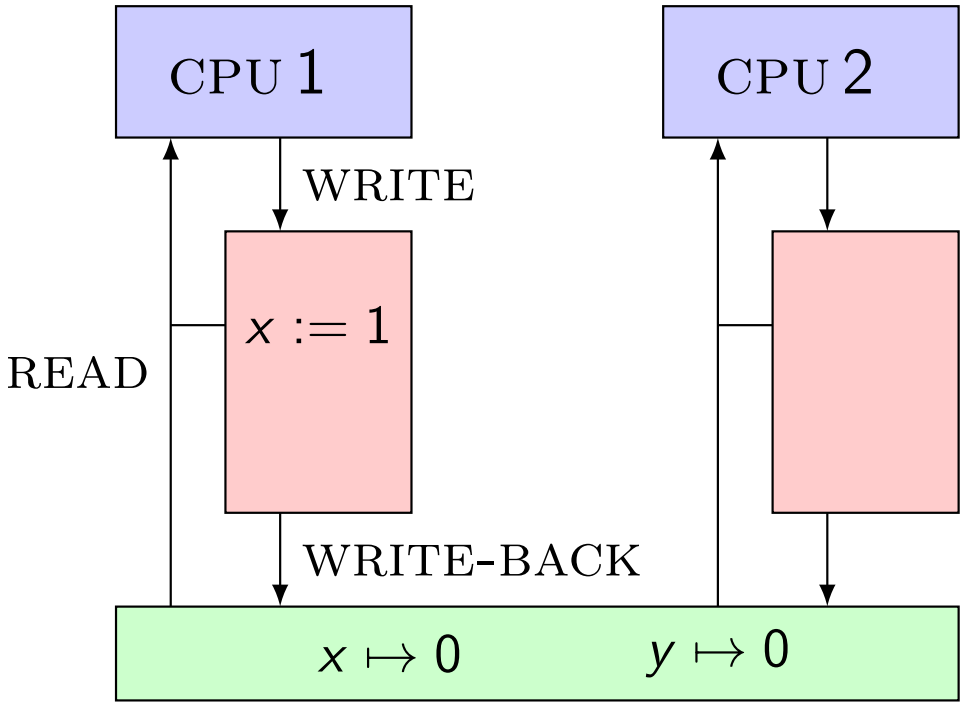
▶ $x := 1;$	▶ $y := 1;$
$a := y;$ // 0	$b := x;$ // 0



Store buffering in x86-TSO

Initially, $x = y = 0$.

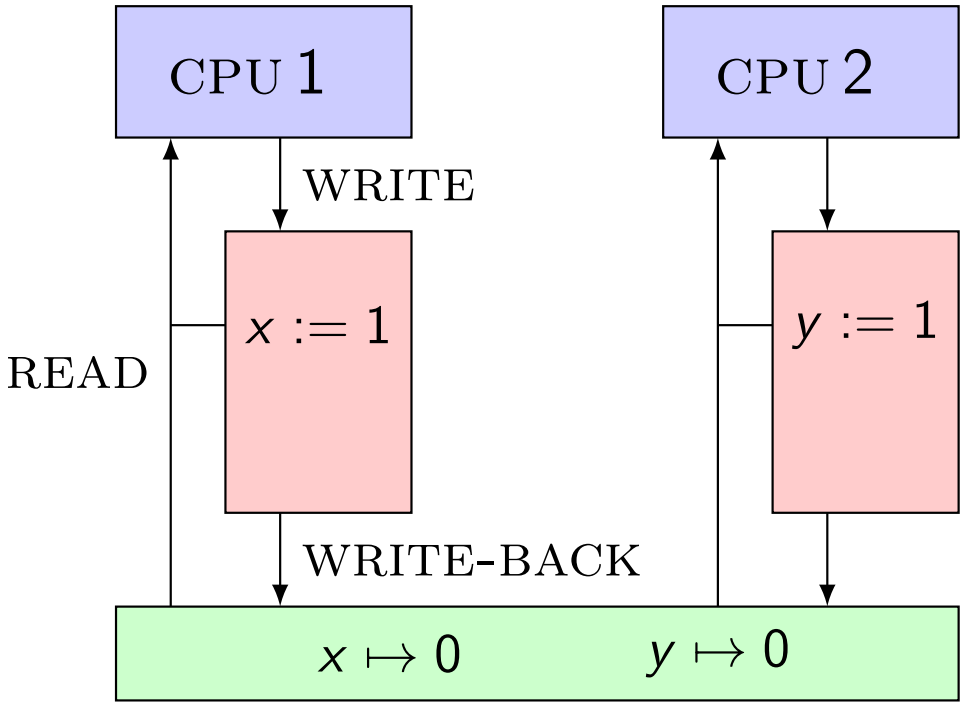
$x := 1;$	$\parallel$	▶ $y := 1;$
▶ $a := y;$ // 0	$\parallel$	$b := x;$ // 0



Store buffering in x86-TSO

Initially, $x = y = 0$.

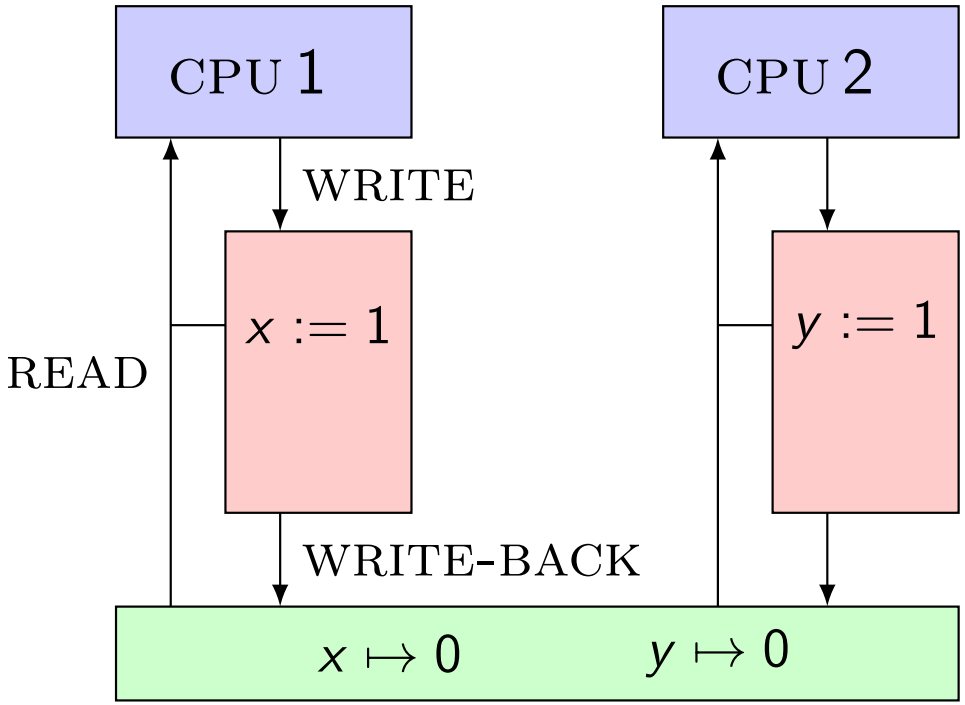
$x := 1;$	$\parallel$	$y := 1;$
▶ $a := y;$ // 0	$\parallel$	▶ $b := x;$ // 0



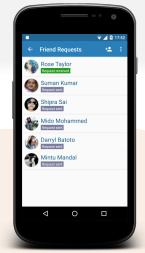
Store buffering in x86-TSO

Initially, $x = y = 0$.

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

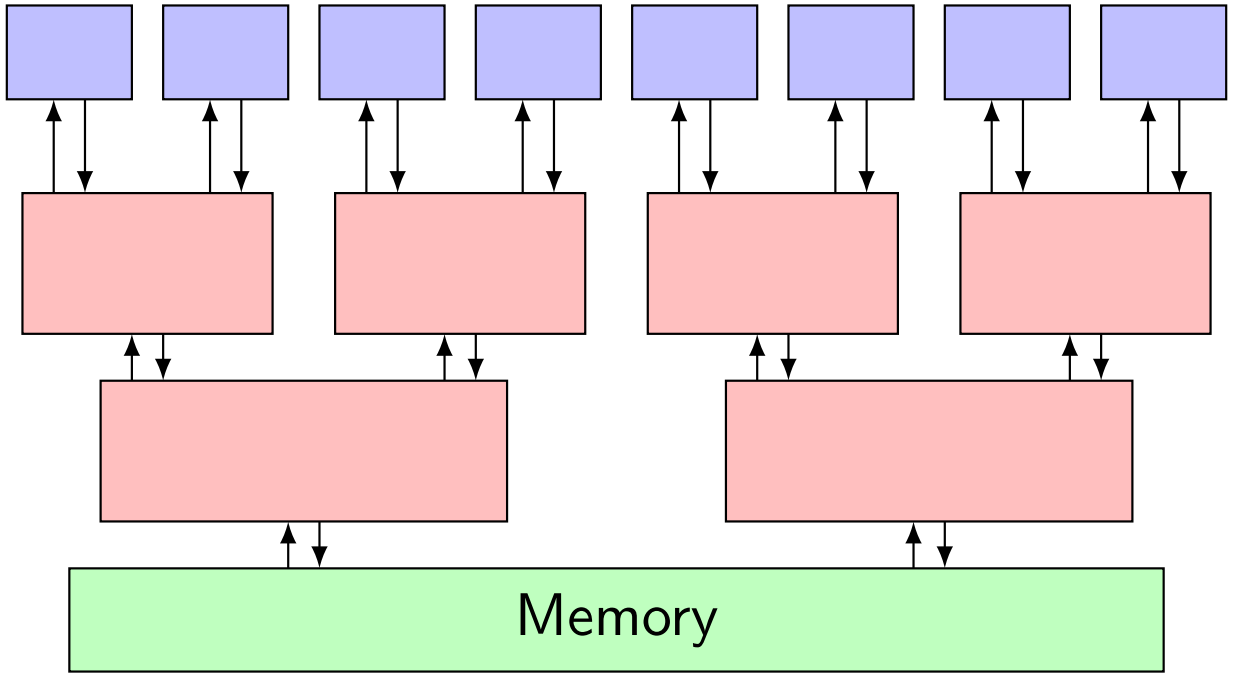


Load buffering in ARM

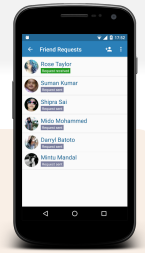


Initially, $x = y = 0$.

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

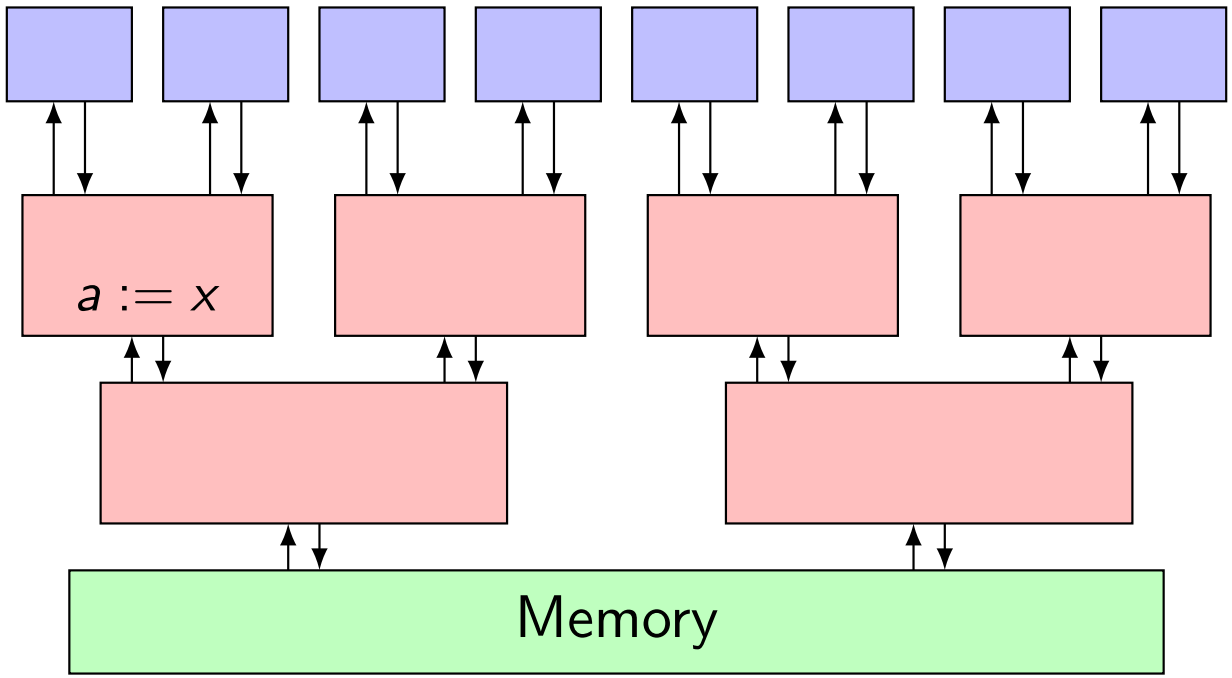


Load buffering in ARM

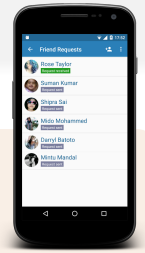


Initially, $x = y = 0$.

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

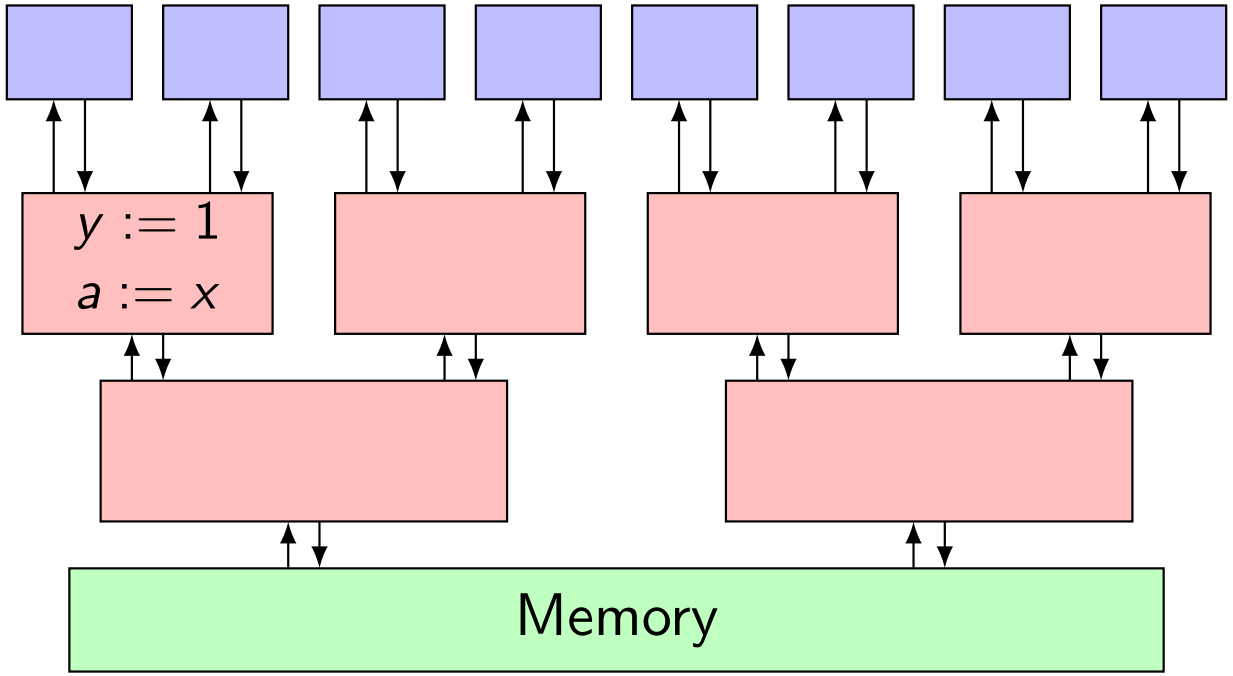


Load buffering in ARM

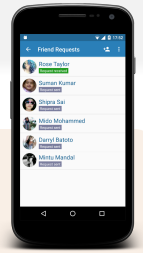


Initially, $x = y = 0$.

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

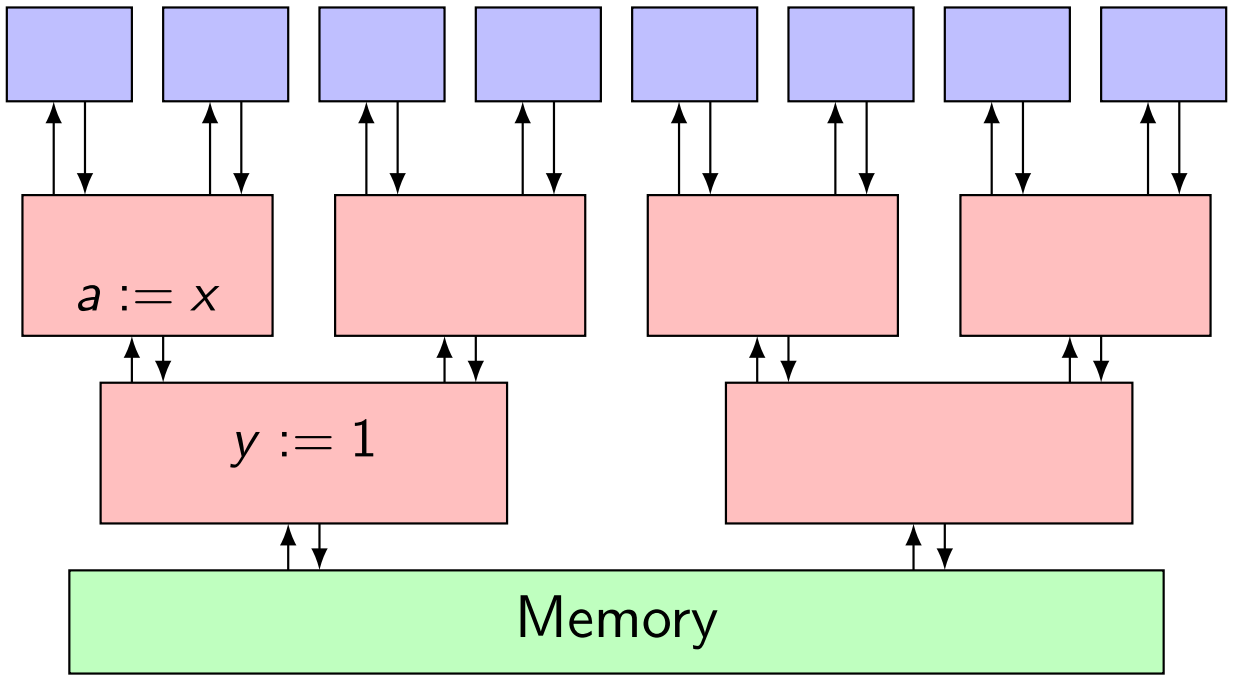


Load buffering in ARM

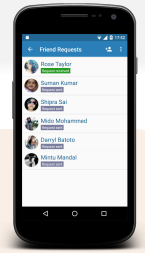


Initially, $x = y = 0$.

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

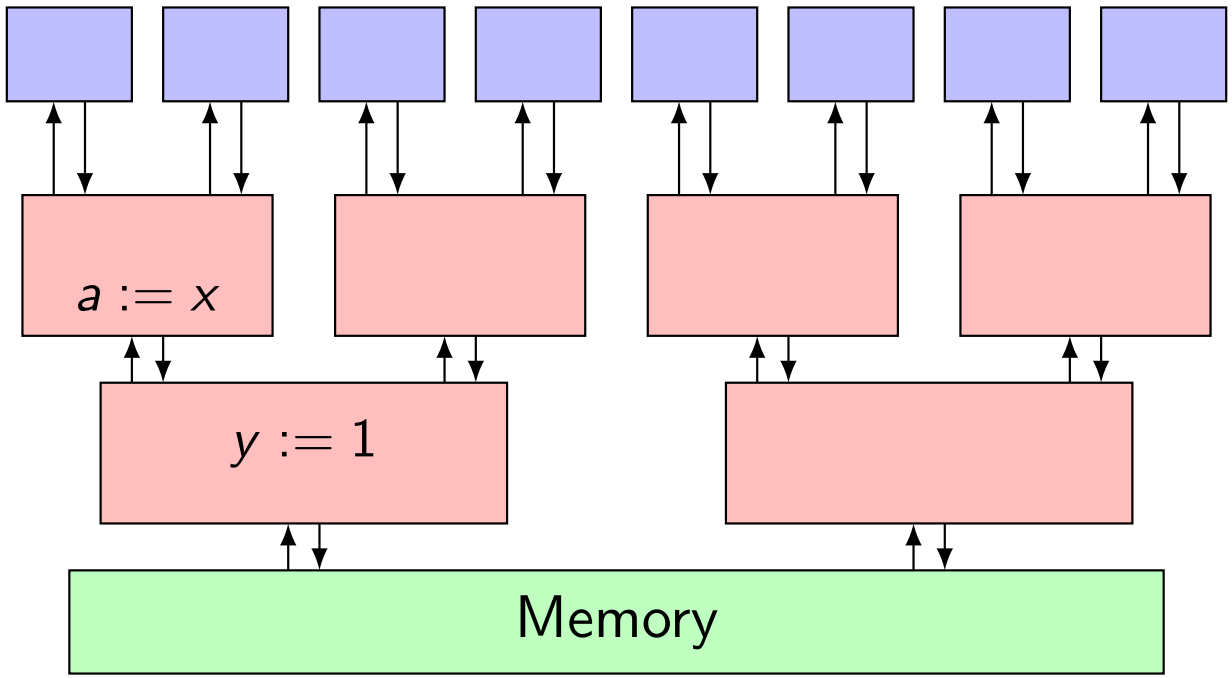


Load buffering in ARM



Initially, $x = y = 0$.

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



Challenge 2: Compilers stir the pot

Initially, $x = y = 0$.

$$\begin{array}{l} x := 1; \\ y := 1; \end{array} \Bigg\| \begin{array}{l} a := x; \\ b := y; \ \ /\!/ \, 1 \\ c := x; \ \ /\!/ \, 0 \end{array}$$

✗ forbidden under SC

Challenge 2: Compilers stir the pot

Initially, $x = y = 0$.

$$\begin{array}{l} x := 1; \\ y := 1; \end{array} \Bigg\| \begin{array}{l} a := x; \\ b := y; \quad /\!/\, 1 \\ c := x; \quad /\!/\, 0 \end{array} \xrightarrow{\textit{compiler optimization}} \begin{array}{l} x := 1; \\ y := 1; \end{array} \Bigg\| \begin{array}{l} a := x; \\ b := y; \quad /\!/\, 1 \\ c := a; \quad /\!/\, 0 \end{array}$$

✗ forbidden under SC

✓ allowed under SC

Challenge 3: Transformations do not suffice

Program transformations fail short to explain some weak behaviors.

- In C/C++:
  - *Release stores* cannot be reordered.
  - *Acquire loads* cannot be reordered.

Message passing (MP)

$$\begin{array}{l|l|l} x :=_{\texttt{rel}} 1; & a := y_{\texttt{acq}}; & /\!/\, 1 \\ y :=_{\texttt{rel}} 1; & b := x_{\texttt{acq}}; & /\!/\, 0 \end{array}$$

Challenge 3: Transformations do not suffice

Program transformations fail short to explain some weak behaviors.

- In C/C++:
  - *Release stores* cannot be reordered.
  - *Acquire loads* cannot be reordered.

Message passing (MP)

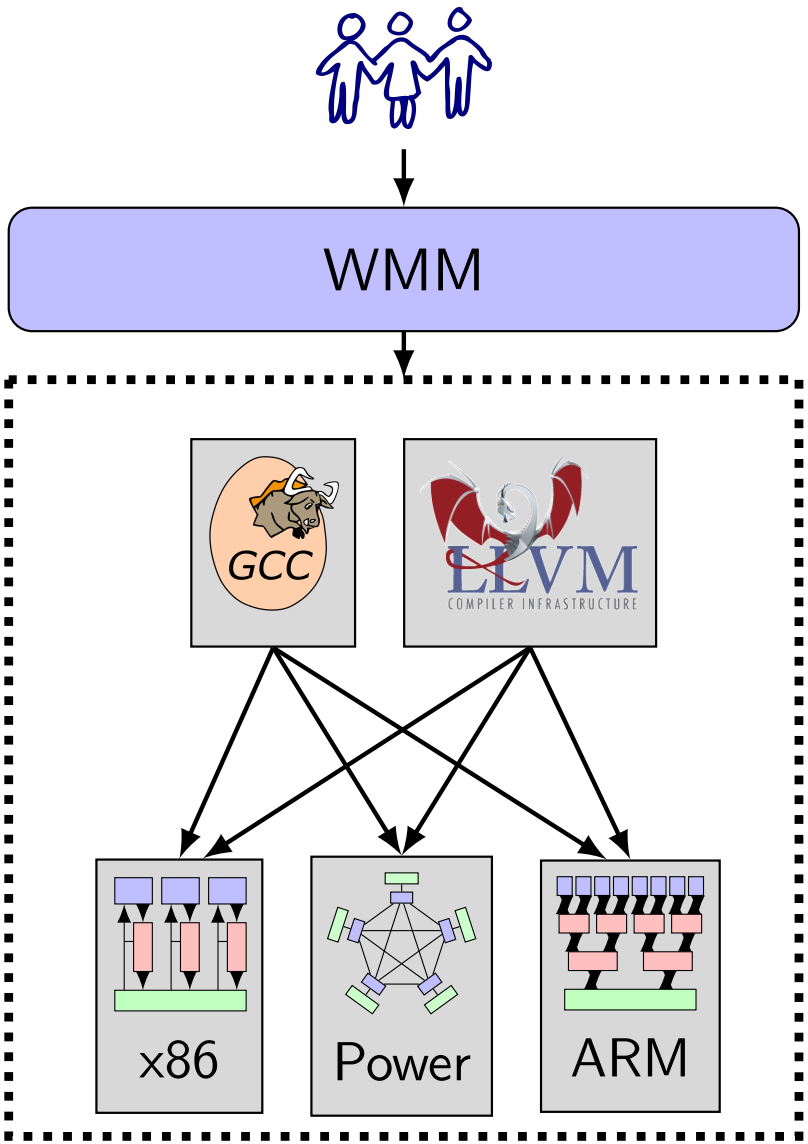
$$\begin{array}{l|l} x :=_{\texttt{rel}} 1; & a := y_{\texttt{acq}}; \ // \ 1 \\ y :=_{\texttt{rel}} 1; & b := x_{\texttt{acq}}; \ // \ 0 \end{array}$$

- And yet, since C/C++ is intended to be compiled to a *non-multi-copy-atomic* architectures:

Independent reads of independent writes (IRIW)

$$\begin{array}{l|l|l|l} a := x_{\texttt{acq}}; \ // \ 1 & & & c := y_{\texttt{acq}}; \ // \ 1 \\ b := y_{\texttt{acq}}; \ // \ 0 & x :=_{\texttt{rel}} 1; & y :=_{\texttt{rel}} 1; & d := x_{\texttt{acq}}; \ // \ 0 \end{array}$$

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)

Implementability

vs.

Programmability

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)	**Atomic**
Races are errors	Welcome to the expert mode

The C11 memory model: Atomics

Two types of accesses

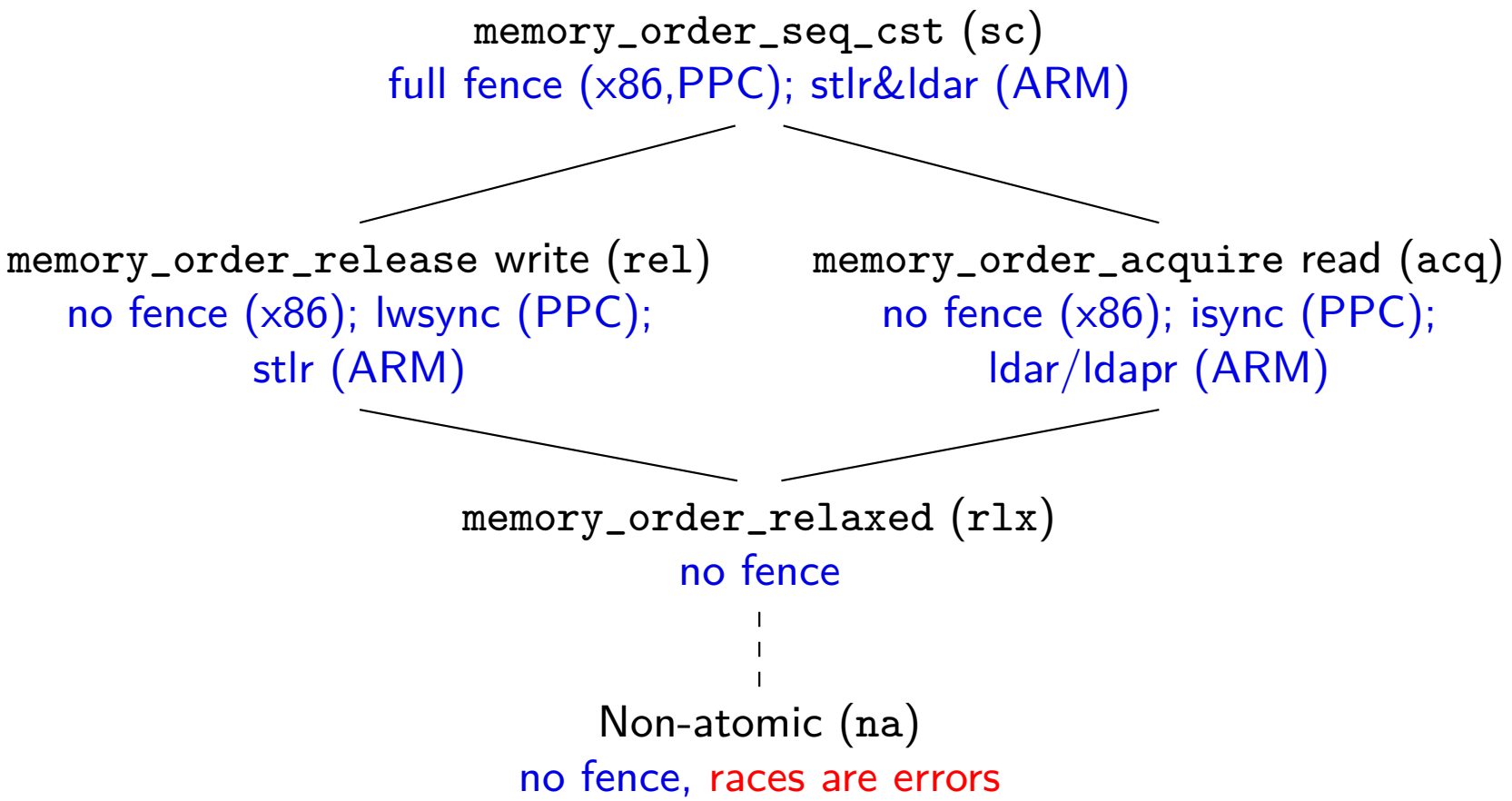
Ordinary (Non-Atomic)	**Atomic**
Races are errors	Welcome to the expert mode

DRF (data race freedom) guarantee

no data races under SC $\Longrightarrow$ only SC behaviors

A spectrum of access modes

`non-atomic` ⊏ `relaxed` ⊏ `release/acquire` ⊏ `sc`



+ Explicit primitives for fences

C11: a declarative memory model

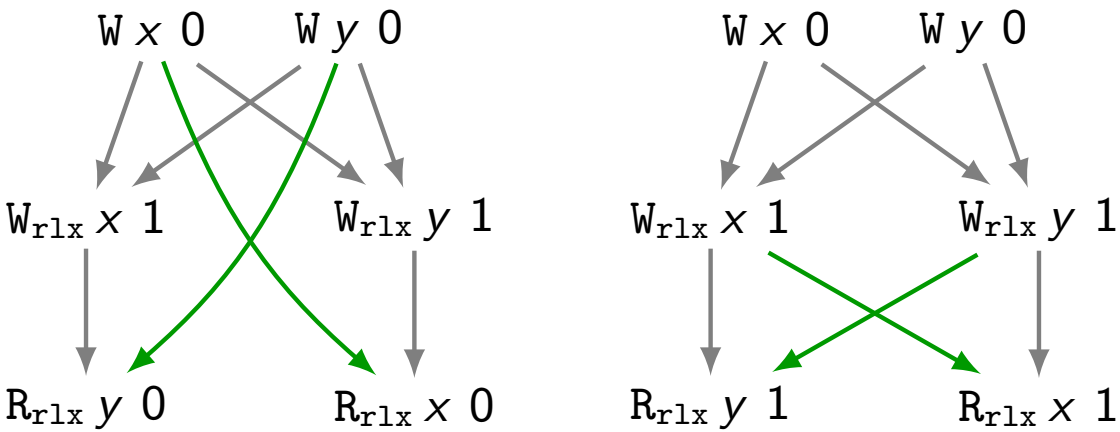
Declarative semantics abstracts away from implementation details.

1. a program $\rightsquigarrow$ a set of directed graphs.
2. The model defines what executions are *consistent*.
3. C/C++11 also has *catch-fire* semantics (forbidden data races).

Execution graphs

Store buffering (SB)

$$x = y = 0$$
$$x :=_{\texttt{rlx}} 1; \;\|\; y :=_{\texttt{rlx}} 1;$$
$$a := y_{\texttt{rlx}}; \;\|\; b := x_{\texttt{rlx}};$$



Relations

- Program order, po
- Reads-from, rf

C/C++11 formal model

[Vafeiadis & Narayan OOPSLA'13]

$$
\begin{array}{rl}
[\![-]\!] & : \ \mathsf{CExp} \to \mathbb{P}(\langle \mathit{res} : \mathsf{Val} \cup \{\bot\}, \mathcal{A} : \mathbb{P}(\mathsf{AName}), \mathsf{lab} : \mathcal{A} \to \mathsf{Act}, \mathsf{sb} : \mathbb{P}(\mathcal{A} \times \mathcal{A}), \mathit{fst} : \mathcal{A}, \mathit{lst} : \mathcal{A}\rangle) \\
[\![v]\!] & \stackrel{\text{def}}{=} \{\langle v, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathsf{skip}\} \\
[\![\mathbf{alloc}()]\!] & \stackrel{\text{def}}{=} \{\langle \ell, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \ell \in \mathsf{Loc} \wedge \mathsf{lab}(a) = \mathrm{A}(\ell)\} \\
[\![[v]_Z := v']\!] & \stackrel{\text{def}}{=} \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathrm{W}_Z(v, v')\} \\
[\![[v]_Z]\!] & \stackrel{\text{def}}{=} \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge v' \in \mathsf{Val} \wedge \mathsf{lab}(a) = \mathrm{R}_Z(v, v')\} \\
[\![\mathbf{CAS}_{X,Y}(v, v_o, v_n)]\!] & \stackrel{\text{def}}{=} \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge v' \in \mathsf{Val} \wedge v' \neq v_o \wedge \mathsf{lab}(a) = \mathrm{R}_Y(v, v')\} \\
& \quad \cup \{\langle v_o, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{AName} \wedge \mathsf{lab}(a) = \mathrm{RMW}_X(v, v_o, v_n)\} \\
[\![\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2]\!] & \stackrel{\text{def}}{=} \{\langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \mid \langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \in [\![E_1]\!]\} \\
& \quad \cup \{\langle \mathit{res}_2, \mathcal{A}_1 \uplus \mathcal{A}_2, \mathsf{lab}_1 \cup \mathsf{lab}_2, \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(\mathit{lst}_1, \mathit{fst}_2)\}, \mathit{fst}_1, \mathit{lst}_2\rangle \mid \\
& \qquad \langle v_1, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \in [\![E_1]\!] \wedge \langle \mathit{res}_2, \mathcal{A}_2, \mathsf{lab}_2, \mathsf{sb}_2, \mathit{fst}_2, \mathit{lst}_2\rangle \in [\![E_2[v_1/x]]\!]\} \\
[\![\mathbf{repeat}\ E\ \mathbf{end}]\!] & \stackrel{\text{def}}{=} \{\langle \mathit{res}_N, \biguplus_{i\in[1..N]} \mathcal{A}_i, \bigcup_{i\in[1..N]} \mathsf{lab}_i, \bigcup_{i\in[1..N]} \mathsf{sb}_i \cup \{(\mathit{lst}_1, \mathit{fst}_2), \ldots, (\mathit{lst}_{N-1}, \mathit{fst}_N)\}, \mathit{fst}_1, \mathit{lst}_N\rangle \mid \\
& \qquad \forall i.\ \langle \mathit{res}_i, \mathcal{A}_i, \mathsf{lab}_i, \mathsf{sb}_i, \mathit{fst}_i, \mathit{lst}_i\rangle \in [\![E]\!] \wedge (i \neq N \implies \mathit{res}_i = 0) \wedge \mathit{res}_N \neq 0\} \\
[\![E_1 \| E_2]\!] & \stackrel{\text{def}}{=} \{\langle \mathsf{combine}(\mathit{res}_1, \mathit{res}_2), \mathcal{A}_1 \uplus \mathcal{A}_2 \uplus \{a_{\text{fork}}, a_{\text{join}}\}, \mathsf{lab}_1 \cup \mathsf{lab}_2 \cup \{a_{\text{fork}} \mapsto \mathsf{skip}, a_{\text{join}} \mapsto \mathsf{skip}\}, \\
& \qquad \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(a_{\text{fork}}, \mathit{fst}_1), (a_{\text{fork}}, \mathit{fst}_2), (\mathit{lst}_1, a_{\text{join}}), (\mathit{lst}_2, a_{\text{join}})\}, a_{\text{fork}}, a_{\text{join}}\rangle \mid \\
& \qquad \langle \mathit{res}_1, \mathcal{A}_1, \mathsf{sb}_1, \mathit{fst}_1, \mathit{lst}_1\rangle \in [\![E_1]\!] \wedge \langle \mathit{res}_2, \mathcal{A}_2, \mathsf{sb}_2, \mathit{fst}_2, \mathit{lst}_2\rangle \in [\![E_2]\!] \wedge a_{\text{fork}}, a_{\text{join}} \in \mathsf{AName}\}
\end{array}
$$

Figure 2. Semantics of closed program expressions.

$$
\begin{array}{rl}
\nexists x.\ \mathsf{hb}(x,x) & \text{(IrreflexiveHB)} \\
\forall \ell.\ \mathsf{totalorder}(\{a \in \mathcal{A} \mid \mathsf{iswrite}_\ell(a)\}, \mathsf{mo}) \wedge \mathsf{hb}_\ell \subseteq \mathsf{mo} & \text{(ConsistentMO)} \\
\mathsf{totalorder}(\{a \in \mathcal{A} \mid \mathsf{isSeqCst}(a)\}, \mathsf{sc}) \wedge \mathsf{hb}_{\mathsf{SeqCst}} \subseteq \mathsf{sc} \wedge \mathsf{mo}_{\mathsf{SeqCst}} \subseteq \mathsf{sc} & \text{(ConsistentSC)} \\
\forall b.\ \mathsf{rf}(b) \neq \bot \iff \exists \ell, a.\ \mathsf{iswrite}_\ell(a) \wedge \mathsf{isread}_\ell(b) \wedge \mathsf{hb}(a,b) & \text{(ConsistentRFdom)} \\
\forall a, b.\ \mathsf{rf}(b) = a \implies \exists \ell, v.\ \mathsf{iswrite}_{\ell,v}(a) \wedge \mathsf{isread}_{\ell,v}(b) \wedge \neg\mathsf{hb}(b,a) & \text{(ConsistentRF)} \\
\forall a, b.\ \mathsf{rf}(b) = a \wedge (\mathsf{mode}(a) = \mathrm{na} \vee \mathsf{mode}(b) = \mathrm{na}) \implies \mathsf{hb}(a,b) & \text{(ConsistentRFna)} \\
\forall a, b.\ \mathsf{rf}(b) = a \wedge \mathsf{isSeqCst}(b) \implies \mathsf{isc}(a,b) \vee \neg\mathsf{isSeqCst}(a) \wedge (\forall x.\ \mathsf{isc}(x,b) \Rightarrow \neg\mathsf{hb}(a,x)) & \text{(RestrSCReads)} \\
\nexists a, b.\ \mathsf{hb}(a,b) \wedge \mathsf{mo}(\mathsf{rf}(b), \mathsf{rf}(a)) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) & \text{(CoherentRR)} \\
\nexists a, b.\ \mathsf{hb}(a,b) \wedge \mathsf{mo}(\mathsf{rf}(b), a) \wedge \mathsf{iswrite}(a) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) & \text{(CoherentWR)} \\
\nexists a, b.\ \mathsf{hb}(a,b) \wedge \mathsf{mo}(b, \mathsf{rf}(a)) \wedge \mathsf{iswrite}(b) \wedge \mathsf{locs}(a) = \mathsf{locs}(b) & \text{(CoherentRW)} \\
\forall a.\ \mathsf{isrmw}(a) \wedge \mathsf{rf}(a) \neq \bot \implies \mathsf{mo}(\mathsf{rf}(a), a) \wedge \nexists c.\ \mathsf{mo}(\mathsf{rf}(a), c) \wedge \mathsf{mo}(c, a) & \text{(AtomicRMW)} \\
\forall a, b, \ell.\ \mathsf{lab}(a) = \mathsf{lab}(b) = \mathrm{A}(\ell) \implies a = b & \text{(ConsistentAlloc)}
\end{array}
$$

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

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

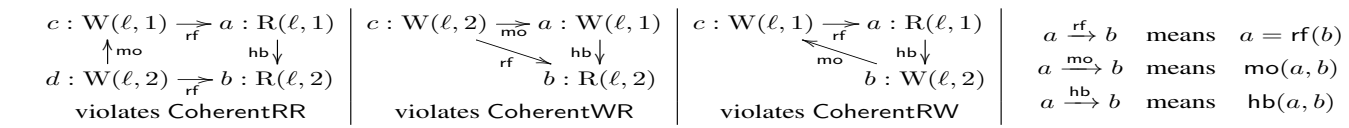


Figure 4. Sample executions violating coherency conditions (Batty et al. 2011).

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

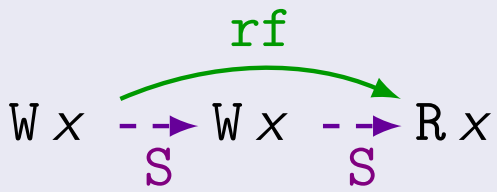
Sequential Consistency (SC)

Definition (Declarative definition of SC, Lamport '79)

G is SC-*consistent* if there exists a relation $\mathtt{S}$ s.t. the following hold:

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

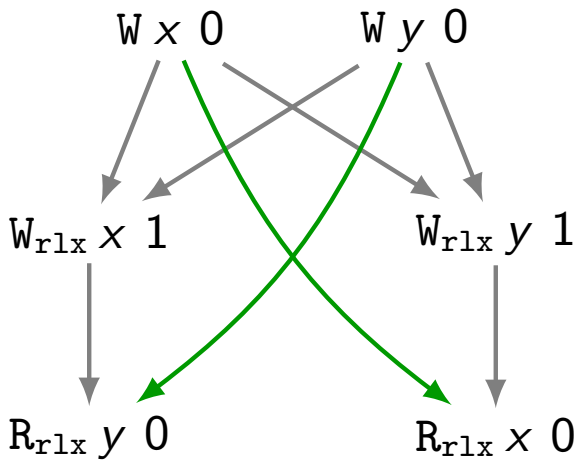
  Namely, the following is *disallowed:*



SC: Example

$$x = y = 0$$

$$x :=_{\texttt{rlx}} 1; \quad a := y_{\texttt{rlx}}; \ /\!/ 0 \quad \Big\| \quad y :=_{\texttt{rlx}} 1; \quad b := x_{\texttt{rlx}}; \ /\!/ 0$$



program order

reads from

not SC-consistent!

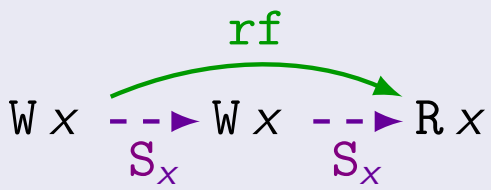
SC-per-location

Definition (SC-per-location)

G is satisfies *SC-per-location* if for every location x, there exists a relation $\mathtt{S}_x$ s.t. the following hold:

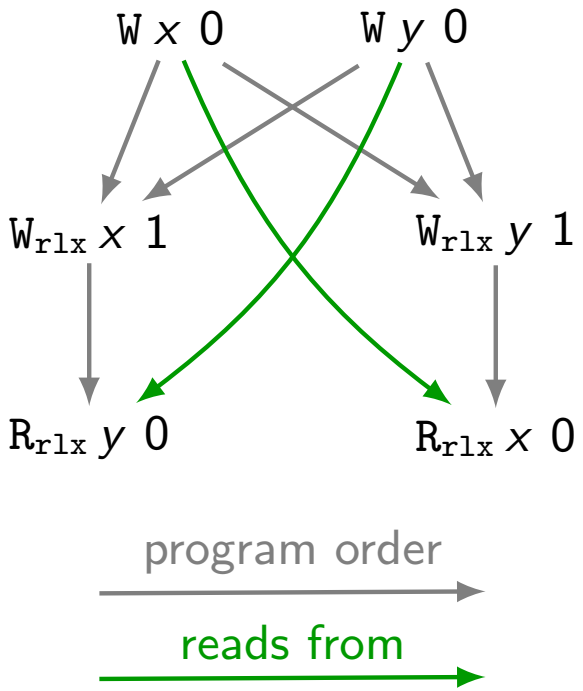
- $\mathtt{S}_x$ is a total order on the events of G that access x.
- If $\mathtt{po} \cup \mathtt{rf} \subseteq \mathtt{S}_x$.
- If $\langle a, b \rangle \in \mathtt{rf}$ then there does not exist $c \in \mathtt{W}_x$ such that $\langle a, c \rangle \in \mathtt{S}_x$ and $\langle c, b \rangle \in \mathtt{S}_x$.

  Namely, the following is *disallowed:*



SC-per-location: Example 1

$$
\begin{array}{c}
x = y = 0 \\
\begin{array}{l|l}
x :=_{\texttt{rlx}} 1; & y :=_{\texttt{rlx}} 1; \\
a := y_{\texttt{rlx}}; \ /\!/ \, 0 & b := x_{\texttt{rlx}}; \ /\!/ \, 0
\end{array}
\end{array}
$$

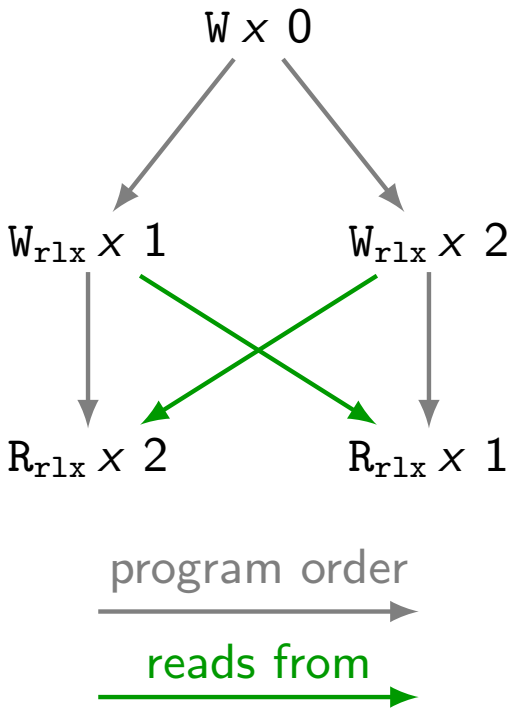


satisfies SC-per-location!

SC-per-location: Example 2

$$x = 0$$

$$x :=_{\texttt{rlx}} 1; \quad a := x_{\texttt{rlx}}; \ /\!/ 2 \ \Big\| \ x :=_{\texttt{rlx}} 2; \quad b := x_{\texttt{rlx}}; \ /\!/ 1$$



does not satisfy SC-per-location!

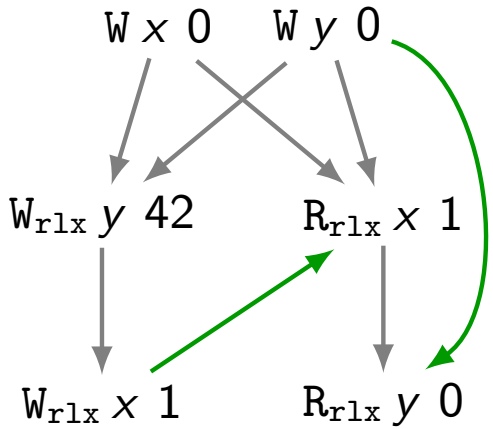
Release/Acquire synchronization

SC-per-location is often *too weak*:

- It does not support the message passing idiom:

Message passing (MP)

$$\begin{array}{l|l} y :=_{\texttt{rlx}} 42; & a := x_{\texttt{rlx}}; \ \#1 \\ x :=_{\texttt{rlx}} 1; & b := y_{\texttt{rlx}}; \ \#0 \end{array}$$



- Also: we cannot implement locks.

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

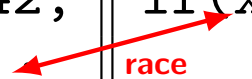
1

```
        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;  ||    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(x_rlx == 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;   ||    print(y);
         ||  }
```

race

2

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

race

Synchronization in C/C++11 through examples

1

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

race

2

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

race

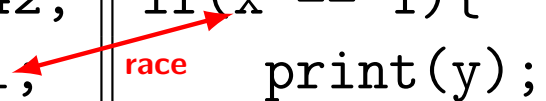
3

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

Synchronization in C/C++11 through examples

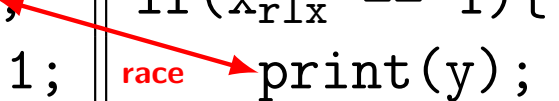
1

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



2

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



3

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_acq == 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;           ||      print(y);
                 ||  }
```


2

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


3

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_acq == 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;           ||      print(y);
                 ||  }
```


2

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


3

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


4

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_rlx == 1){
fence_rel;       ||      fence_acq;
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;           ||      print(y);
                 ||  }
```


2

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


3

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


4

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_rlx == 1){
fence_rel;       ||      fence_acq;
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;           ||      print(y);
                 ||  }
```

race

2

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

race

3

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

rf, sw

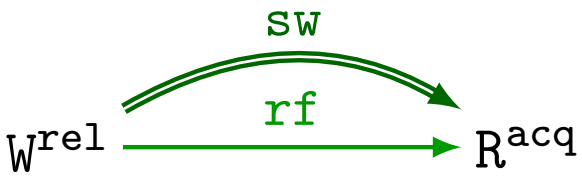
4

```
int y = 0;
atomic<int> x = 0;
y = 42;          ||  if(x_rlx == 1){
fence_rel;       ||      fence_acq;
x =_rlx 1;       ||      print(y);
                 ||  }
```

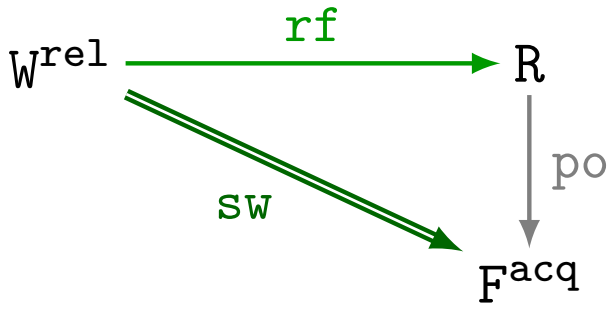
rf, sw

The “synchronizes-with” relation

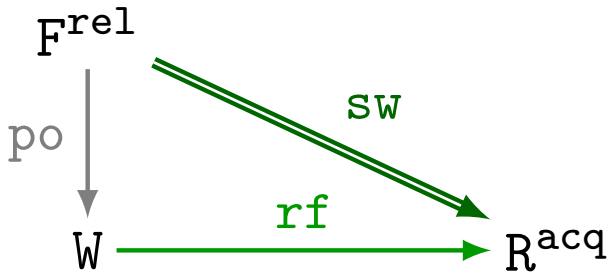
```
y = 42;          ||  if(x_acq == 1){
x =_rel 1;       ||      print(y);
                 ||  }
```



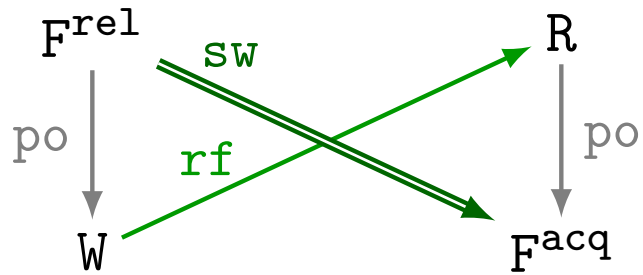

```
y = 42;          ||  if(x_rlx == 1){
x =_rel 1;       ||      fence_acq;
                 ||      print(y);
                 ||  }
```



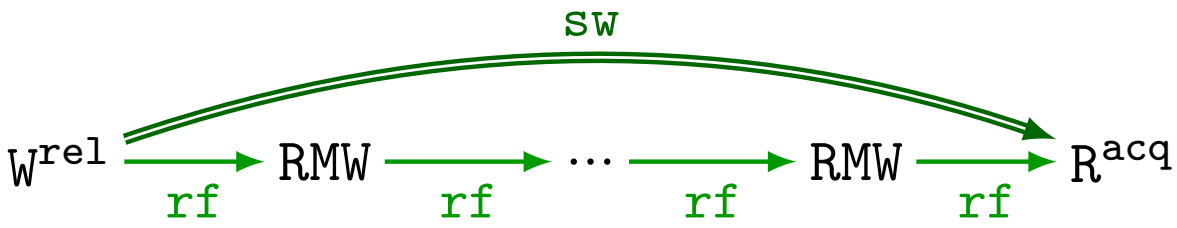

```
y = 42;          ||  if(x_acq == 1){
fence_rel;       ||      print(y);
x =_rlx 1;       ||  }
```

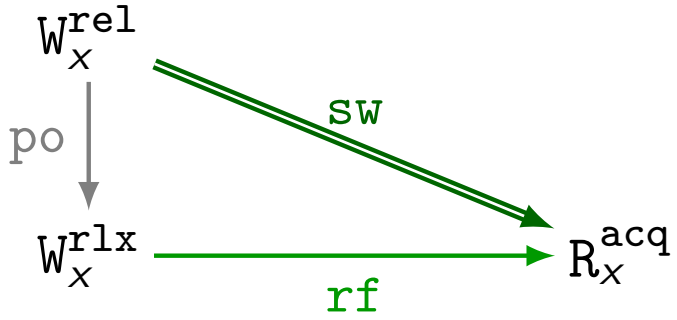



```
y = 42;          ||  if(x_rlx == 1){
fence_rel;       ||      fence_acq;
x =_rlx 1;       ||      print(y);
                 ||  }
```



The "synchronizes-with" relation: Release sequences





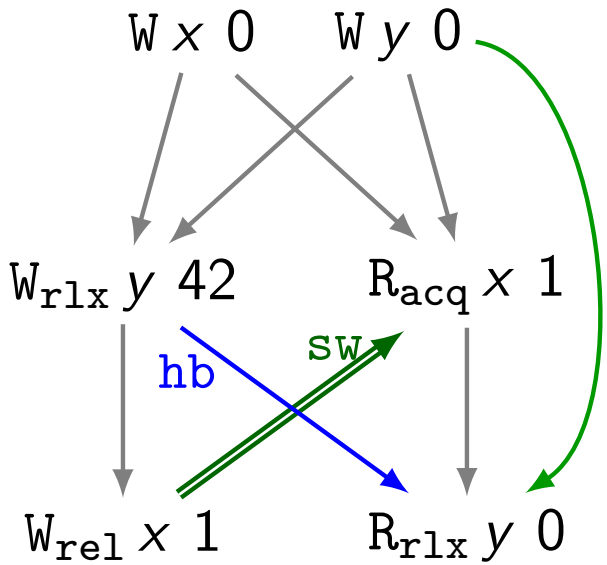
- Note: the latter case will be deprecated in C++20.

The "happens-before" relation

Definition (happens-before)

$$\frac{a \xrightarrow{\text{po}} b}{a \xrightarrow{\text{hb}} b} \qquad \frac{a \xRightarrow{\text{sw}} b}{a \xrightarrow{\text{hb}} b} \qquad \frac{a \xrightarrow{\text{hb}} b \qquad b \xrightarrow{\text{hb}} c}{a \xrightarrow{\text{hb}} c}$$

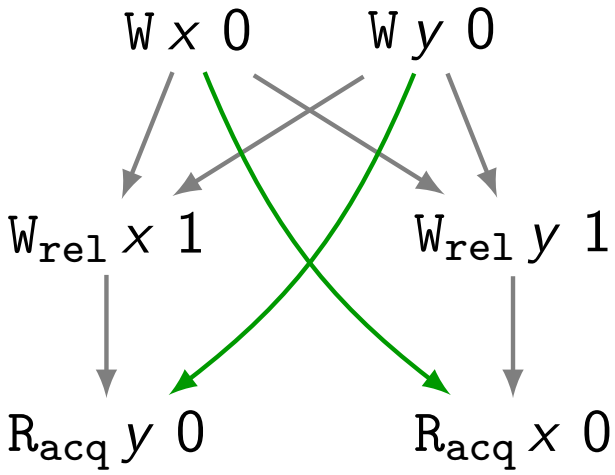
- hb should be acyclic.
- The SC-per-location orders should never contradict hb.



SC accesses and fences

Store buffer

$x :=_{\texttt{rel}} 1;$	$\Vert$	$y :=_{\texttt{rel}} 1;$
$a := y_{\texttt{acq}};$ // 0	$\Vert$	$b := x_{\texttt{acq}};$ // 0

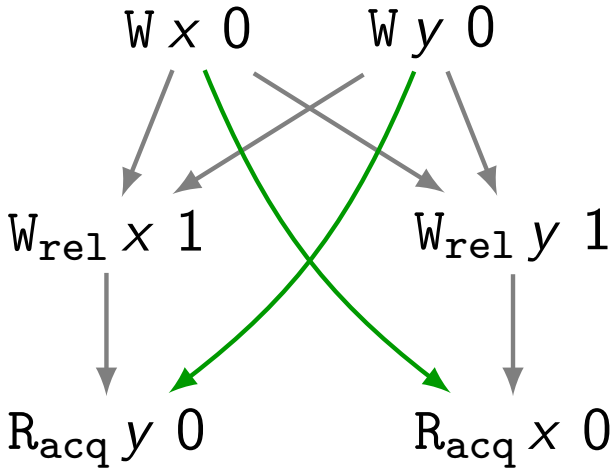


How to guarantee only SC behaviors (*i.e.*, $a = 1 \vee b = 1$)?

SC accesses and fences

Store buffer

$$x :=_{\texttt{rel}} 1; \quad a := y_{\texttt{acq}}; \ // \ 0 \quad \Big\| \quad y :=_{\texttt{rel}} 1; \quad b := x_{\texttt{acq}}; \ // \ 0$$



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

$$x :=_{\texttt{sc}} 1; \quad a := y_{\texttt{sc}}; \quad \Big\| \quad y :=_{\texttt{sc}} 1; \quad b := x_{\texttt{sc}}; \quad \cong \quad x :=_{\texttt{rlx}} 1; \quad \textbf{fence}_{\texttt{sc}}; \quad a := y_{\texttt{rlx}}; \quad \Big\| \quad y :=_{\texttt{rlx}} 1; \quad \textbf{fence}_{\texttt{sc}}; \quad b := x_{\texttt{rlx}};$$

SC semantics

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

$$\begin{array}{l} a := x_{\mathtt{acq}};\ \#\,1 \\ b := y_{\mathtt{sc}};\ \#\,0 \end{array} \;\Big\|\; x :=_{\mathtt{sc}} 1; \;\Big\|\; y :=_{\mathtt{sc}} 1; \;\Big\|\; \begin{array}{l} c := y_{\mathtt{acq}};\ \#\,1 \\ d := x_{\mathtt{sc}};\ \#\,0 \end{array}$$

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

$$\begin{array}{l} a := x_{\mathtt{acq}};\ \#\,1 \\ \mathbf{fence}_{\mathtt{sc}}; \\ b := y_{\mathtt{acq}};\ \#\,0 \end{array} \;\Big\|\; x :=_{\mathtt{rel}} 1; \;\Big\|\; y :=_{\mathtt{rel}} 1; \;\Big\|\; \begin{array}{l} c := y_{\mathtt{acq}};\ \#\,1 \\ \mathbf{fence}_{\mathtt{sc}}; \\ d := x_{\mathtt{acq}};\ \#\,0 \end{array}$$

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

C/C++11 is too weak

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

Load-buffering

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

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

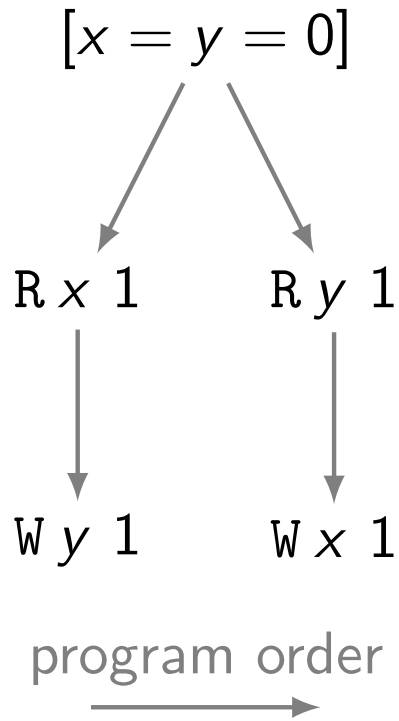
C/C++11 is too weak

`non-atomic` ⊏ `relaxed` ⊏ `release/acquire` ⊏ `sc`

Load-buffering

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

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



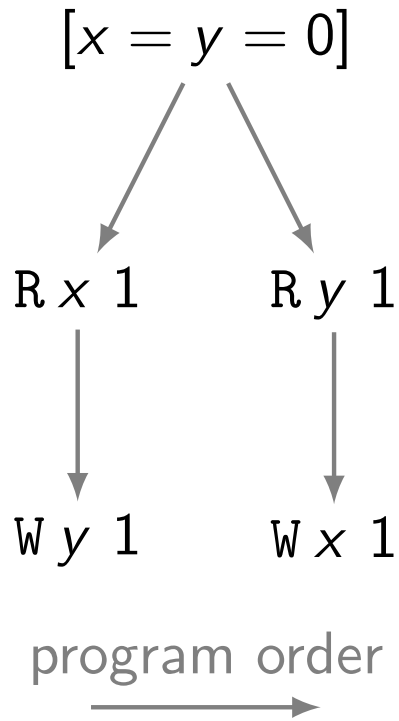
C/C++11 is too weak

`non-atomic` ⊏ `relaxed` ⊏ `release/acquire` ⊏ `sc`

Load-buffering

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

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



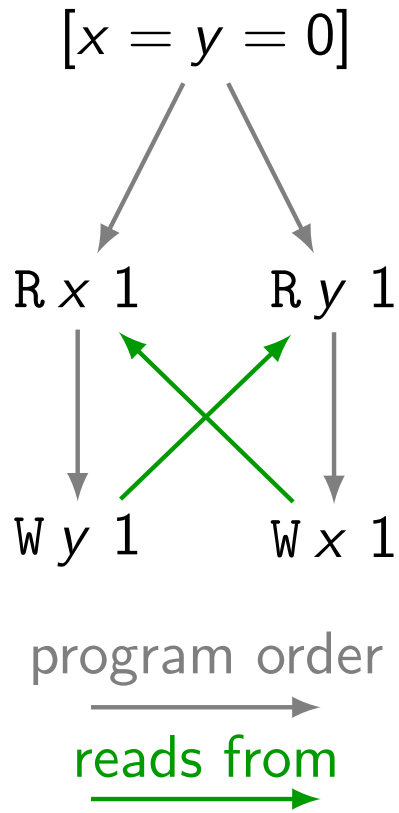
C/C++11 is too weak

`non-atomic` ⊏ `relaxed` ⊏ `release/acquire` ⊏ `sc`

Load-buffering

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

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



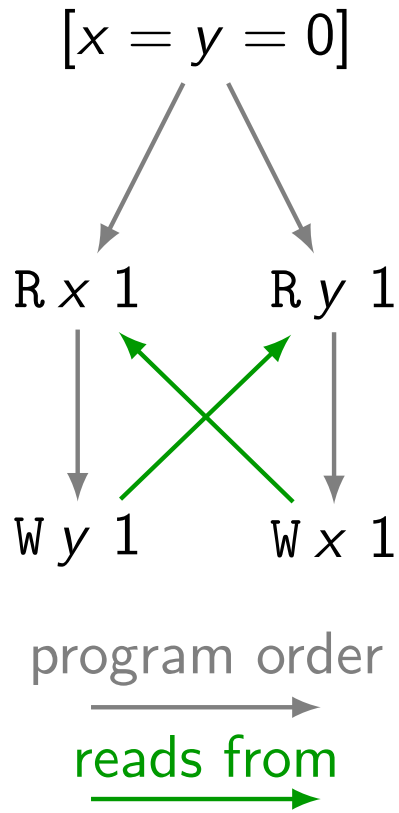
C/C++11 is too weak

`non-atomic` ⊏ `relaxed` ⊏ `release/acquire` ⊏ `sc`

Load-buffering

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

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



C/C++11 is too weak

`non-atomic` ⊏ `relaxed` ⊏ `release/acquire` ⊏ `sc`

Load-buffering

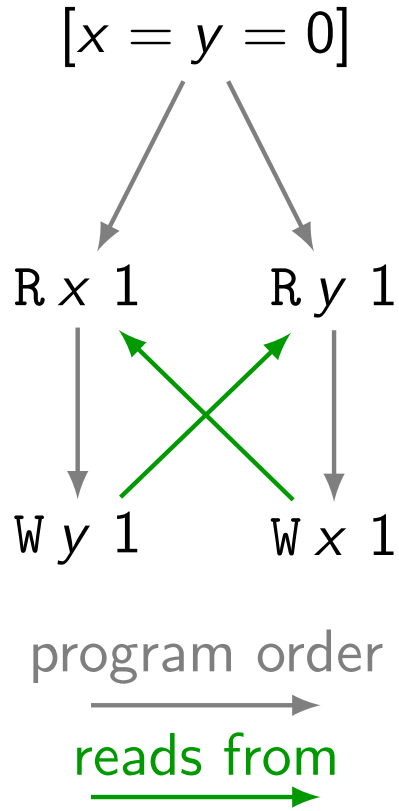
$a := x;$	// 1	$b := y;$	// 1
$y := 1;$		$x := b;$	

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

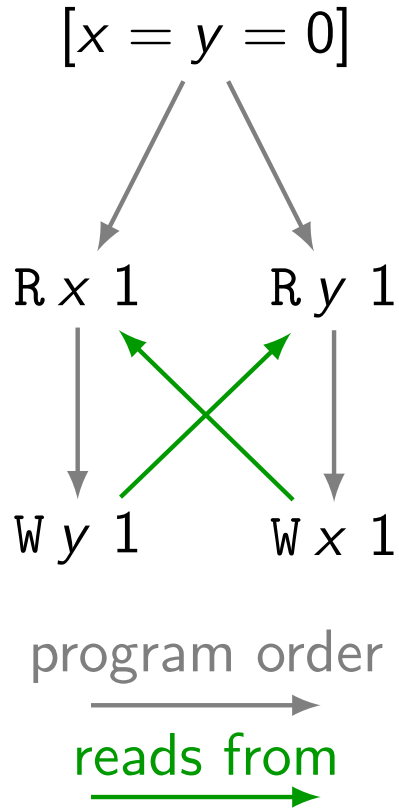
`non-atomic` ⊏ `relaxed` ⊏ `release/acquire` ⊏ `sc`

Load-buffering + control dependency

$a := x;$ // 1	‖	$b := y;$ // 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

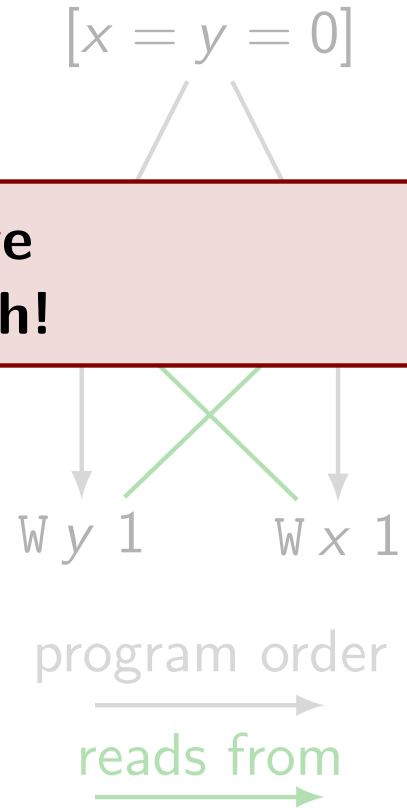
Load-buffering + control dependency

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

The three examples have the same execution graph!

C/

The DRF guarantee is broken!



The hardware solution

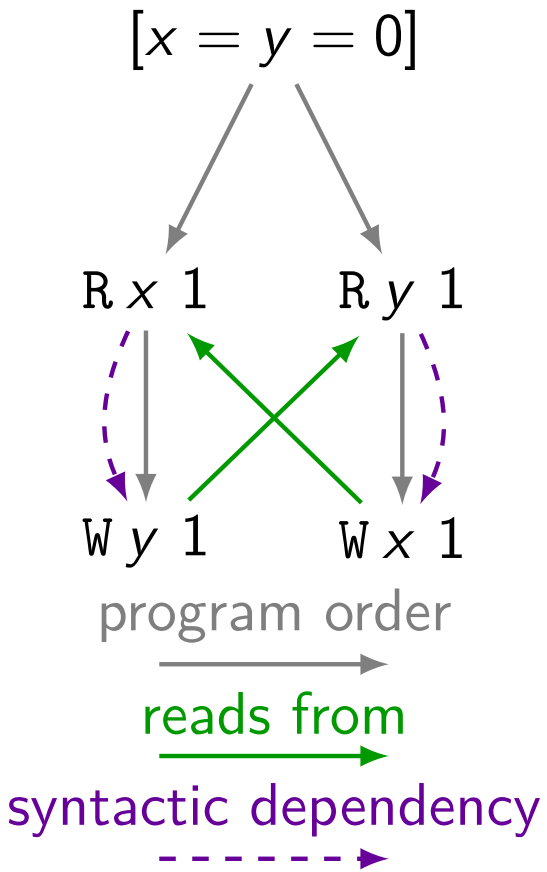
Keep track of syntactic dependencies and forbid dependency cycles.

Load-buffering

$$a := x; \ /\!/ \ 1 \quad \| \quad b := y; \ /\!/ \ 1$$
$$y := 1; \quad \| \quad x := b;$$

Load-buffering + data dependency

$$a := x; \ /\!/ \ 1 \quad \| \quad b := y; \ /\!/ \ 1$$
$$y := a; \quad \| \quad x := b;$$



The hardware solution

Keep track of syntactic dependencies and forbid dependency cycles.

Load-buffering

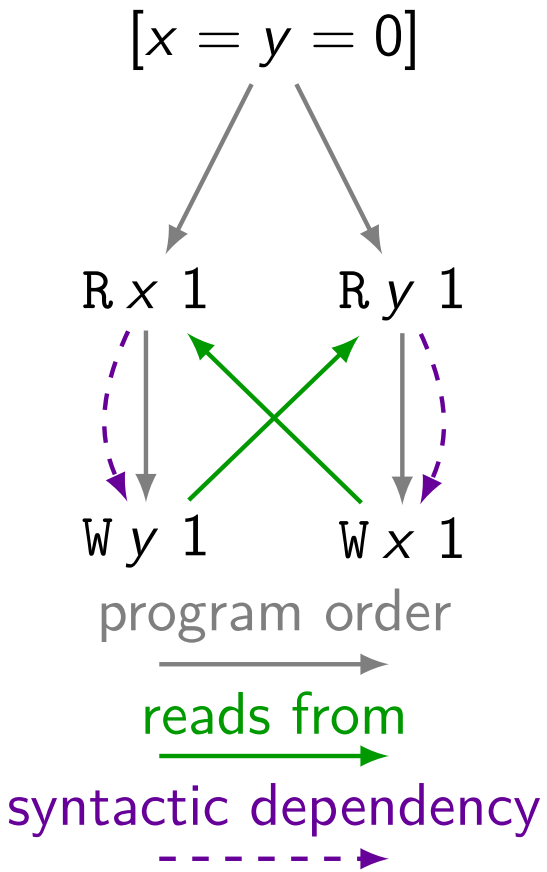
$$\begin{array}{l} a := x; \ \#\!/ 1 \\ y := 1; \end{array} \quad \Big\| \quad \begin{array}{l} b := y; \ \#\!/ 1 \\ x := b; \end{array}$$

Load-buffering + data dependency

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

Load-buffering + fake dependency

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



The hardware solution

Keep track of syntactic dependencies and forbid dependency cycles.

Load-buffering

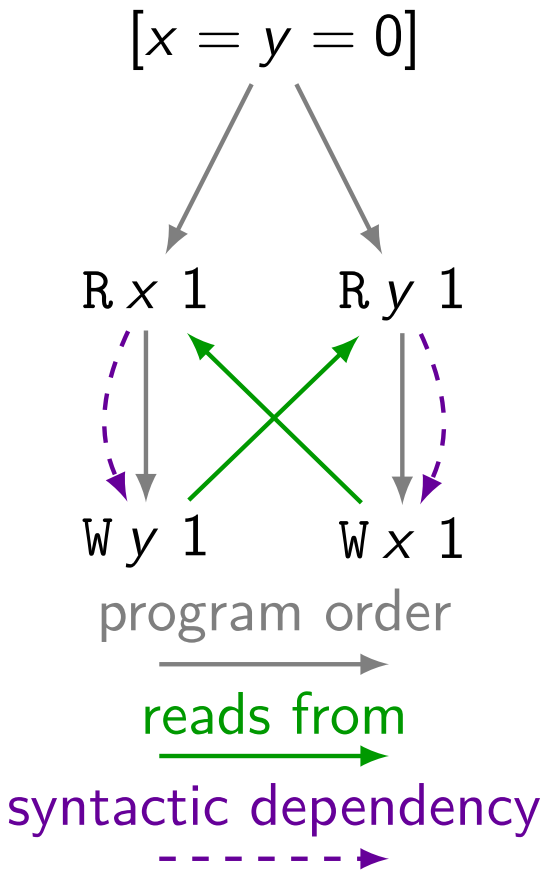
$$a := x; \ /\!/ 1 \quad \| \quad b := y; \ /\!/ 1$$
$$y := 1; \quad \| \quad x := b;$$

Load-buffering + data dependency

$$a := x; \ /\!/ 1 \quad \| \quad b := y; \ /\!/ 1$$
$$y := a; \quad \| \quad x := b;$$

Load-buffering + fake dependency

$$a := x; \ /\!/ 1 \quad \| \quad b := y; \ /\!/ 1$$
$$y := a + 1 - a; \quad \| \quad x := b;$$



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

The “out-of-thin-air” problem

C/C++11 is too weak

- Values might appear *out-of-thin-air*.
- The *DRF guarantee* is broken.

The C++14 standard states:

> *“Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation.”*

- “Defined” by examples.

Solution

A straightforward solution

- Disallow $\texttt{po} \cup \texttt{rf}$ cycles!
- On weak hardware it carries a certain *implementation cost*.

  [Ou and Demsky. Towards understanding the costs of avoiding out-of-thin-air results. OOPSLA'18]

Solution

A straightforward solution

- Disallow $\texttt{po} \cup \texttt{rf}$ cycles!
- On weak hardware it carries a certain *implementation cost*.

  [Ou and Demsky. Towards understanding the costs of avoiding out-of-thin-air results. OOPSLA'18]

RC11 (Repaired C11) model [L, Vafeiadis, Kang, Hur, Dreyer. PLDI'17]

- (Modified) compilation schemes are correct.
- DRF holds and no OOTA-values.
- Model checking [Kokologiannakis, L, Sagonas, Vafeiadis. POPL'18]

  http://plv.mpi-sws.org/rcmc/

Solution

A straightforward solution

- Disallow $\texttt{po} \cup \texttt{rf}$ cycles!
- On weak hardware it carries a certain *implementation cost*.

  [Ou and Demsky. Towards understanding the costs of avoiding out-of-thin-air results. OOPSLA'18]

RC11 (Repaired C11) model [L, Vafeiadis, Kang, Hur, Dreyer. PLDI'17]

- (Modified) compilation schemes are correct.
- DRF holds and no OOTA-values.
- Model checking [Kokologiannakis, L, Sagonas, Vafeiadis. POPL'18]

  http://plv.mpi-sws.org/rcmc/

- Solving the problem without changing the compilation schemes will require a major revision of the standard.

A 'promising' solution to OOTA

[Kang, Hur, L, Vafeiadis, Dreyer POPL'17]

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

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

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

Simple operational semantics for C11's relaxed accesses

Store-buffering

$x = y = 0$

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

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

T_1's view

x	y
0	0

T_2's view

x	y
0	0

- Global memory is a pool of messages of the form

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

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

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

$x = 1;$	$\blacktriangleright\ y = 1;$
$\blacktriangleright\ a = y;\ \ /\!/\ 0$	$b = x;\ \ /\!/\ 0$

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@5 \rangle$

T_1's view

x	y
~~0~~ 5	0

T_2's view

x	y
0	0

- Global memory is a pool of messages of the form

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

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

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle y : 1@5 \rangle$

T_1's view

x	y
~~0~~ 5	0

T_2's view

x	y
0	~~0~~ 5

- Global memory is a pool of messages of the form

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

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

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle y : 1@5 \rangle$

T_1's view

x	y
~~0~~ 5	0

T_2's view

x	y
0	~~0~~ 5

- Global memory is a pool of messages of the form

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

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

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle y : 1@5 \rangle$

T_1's view

x	y
~~0~~	0
5	

T_2's view

x	y
0	~~0~~
	5

- Global memory is a pool of messages of the form

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

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

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle y : 1@5 \rangle$

T_1's view

x	y
~~0~~	0
5	

T_2's view

x	y
0	~~0~~
	5

Coherence Test

$$x = 0$$

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

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle y : 1@5 \rangle$

T_1's view

x	y
~~0~~	0
5	

T_2's view

x	y
0	~~0~~
	5

Coherence Test

$$x = 0$$

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

Memory

$\langle x : 0@0 \rangle$

T_1's view

x
0

T_2's view

x
0

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle y : 1@5 \rangle$

T_1's view

x	y
~~0~~	0
5	

T_2's view

x	y
0	~~0~~
	5

Coherence Test

$$x = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle x : 1@5 \rangle$

T_1's view

x
~~0~~
5

T_2's view

x
0

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle y : 1@5 \rangle$

T_1's view

x	y
~~0~~	0
5	

T_2's view

x	y
0	~~0~~
	5

Coherence Test

$$x = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle x : 2@7 \rangle$

T_1's view

x
~~0~~
5

T_2's view

x
~~0~~
7

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle y : 1@5 \rangle$

T_1's view

x	y
~~0~~	0
5	

T_2's view

x	y
0	~~0~~
	5

Coherence Test

$$x = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle x : 2@7 \rangle$

T_1's view

x
~~0~~
~~5~~
7

T_2's view

x
~~0~~
7

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle y : 1@5 \rangle$

T_1's view

x	y
~~0~~	0
5	

T_2's view

x	y
0	~~0~~
	5

Coherence Test

$$x = 0$$

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

Memory

$\langle x : 0@0 \rangle$
$\langle x : 1@5 \rangle$
$\langle x : 2@7 \rangle$

T_1's view

x
~~0~~
~~5~~
7

T_2's view

x
~~0~~
7

Promises

Load-buffering

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

- 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 $y := 1;$	▶ $x := y;$

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

T_1's view

x	y
0	0

T_2's view

x	y
0	0

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

Promises

Load-buffering

▶ $a := x;$ // 1 $y := 1;$	▶ $x := y;$

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@5 \rangle$

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 $y := 1;$	▶ $x := y;$

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@5 \rangle$

T_1's view

x	y
0	0

T_2's view

x	y
0	~~0~~
	5

- 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

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@5 \rangle$

$\langle x : 1@5 \rangle$

T_1's view

x	y
0	0

T_2's view

x	y
~~0~~	~~0~~
5	5

- 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

$\langle x : 0@0 \rangle$
$\langle y : 0@0 \rangle$
$\langle y : 1@5 \rangle$
$\langle x : 1@5 \rangle$

T_1's view

x	y
~~0~~	0
5	

T_2's view

x	y
~~0~~	~~0~~
5	5

- 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

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@5 \rangle$

$\langle x : 1@5 \rangle$

T_1's view

x	y
~~0~~	~~0~~
5	5

T_2's view

x	y
~~0~~	~~0~~
5	5

- 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

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

Load-buffering + dependency

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

Memory

$\langle x : 0@0 \rangle$

$\langle y : 0@0 \rangle$

$\langle y : 1@5 \rangle$

$\langle x : 1@5 \rangle$

T_1's view

x	y
~~0~~	~~0~~
5	5

T_2's view

x	y
~~0~~	~~0~~
5	5

Must not admit the same execution!

Promises

Load-buffering

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

Load-buffering + dependency

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

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

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

Load buffering + fake dependency

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

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

Load buffering + dependency

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

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

Quiz I

Is this behavior possible?

$a := x;$ // 1
$x := 1;$

Quiz II

Is this behavior possible?

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

Quiz III (due to Hans Boehm)

Is this behavior possible?

$a := x_{\texttt{rlx}}$; // 42	$b := y_{\texttt{rlx}}$;
$y :=_{\texttt{rlx}} a$;	**if** $(b = 42)$
	$\quad c :=$ "*if*";
	else
	$\quad c :=$ "*else*";
	$\quad b := 42$;
	$x :=_{\texttt{rlx}} a$;
	print (c); // prints "*if*"

The full model (POPL'17)

We have extended this basic idea to handle:

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

Results

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

The full model (POPL'17)

We have extended this basic idea to handle:

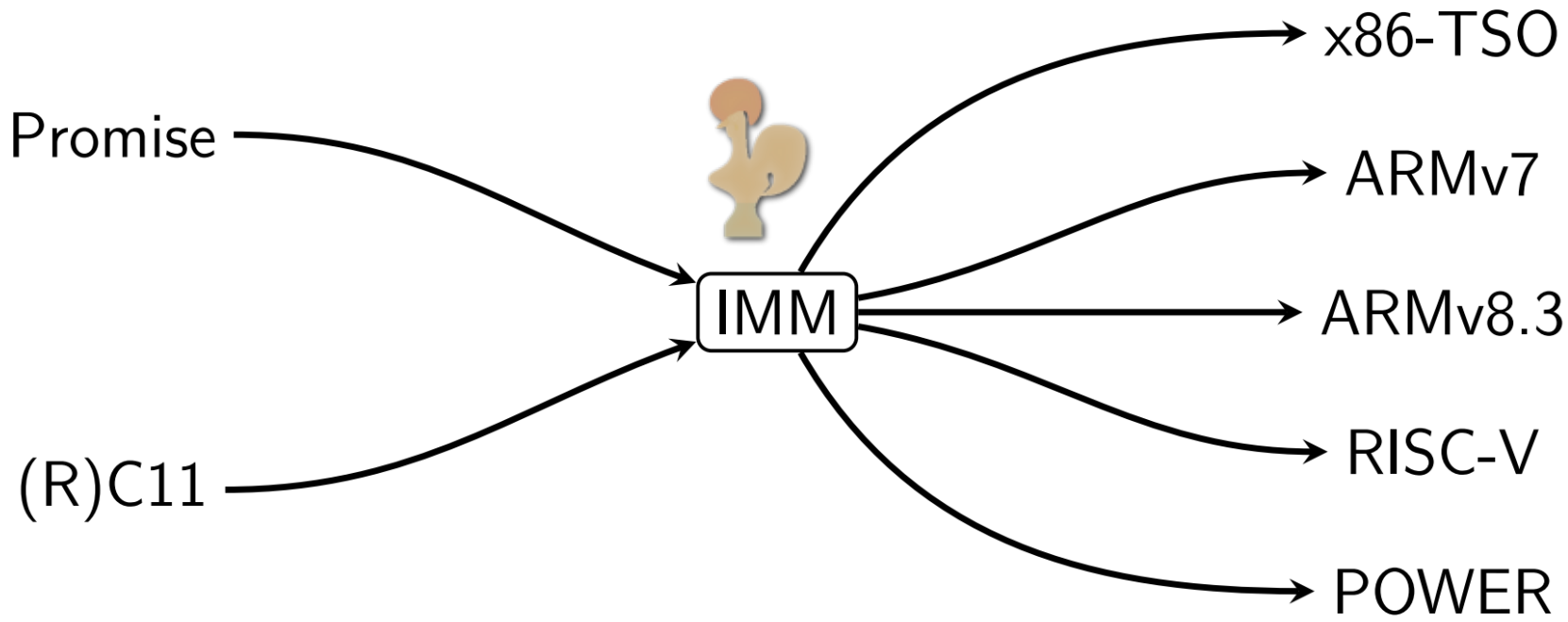
- Atomic updates (e.g., CAS, fetch-and-add)
- Release/acquire fences and accesses
- Release sequences
- SC fences
- Plain accesses
  (C11's non-atomics & Java's normal accesses)

The **Coq** proof assistant

Results

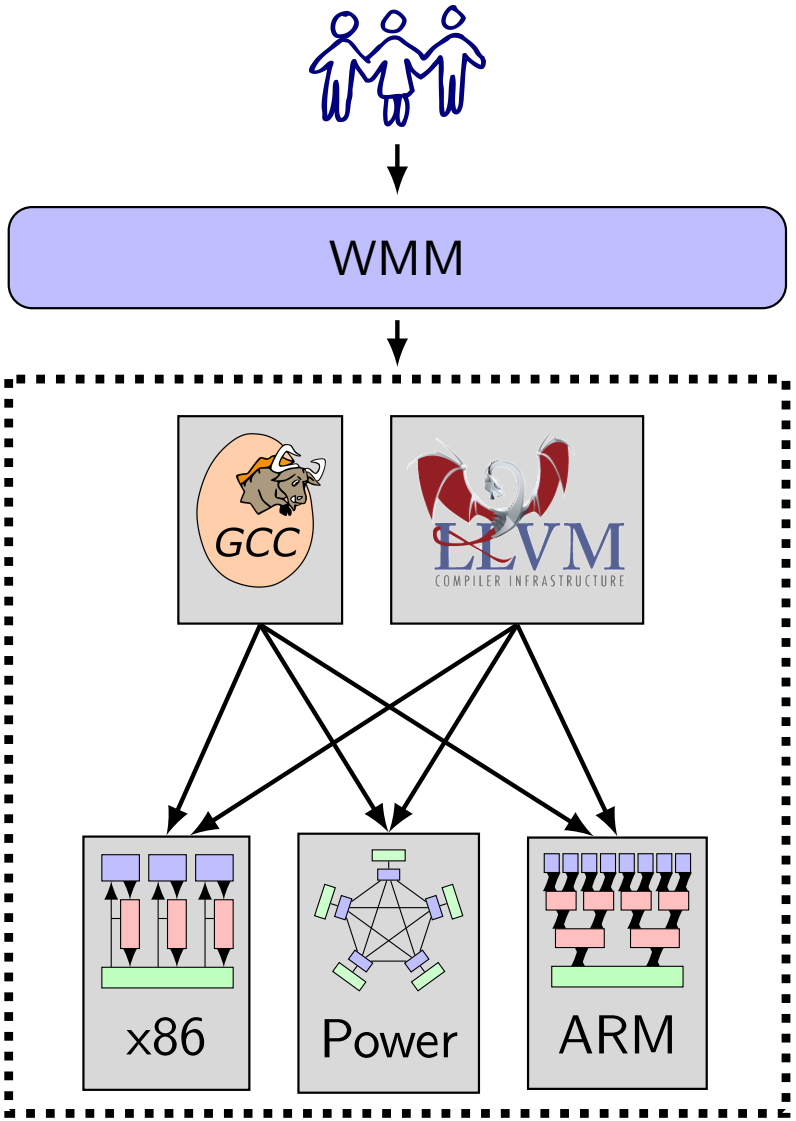
- No "out-of-thin-air" values
- DRF guarantees
- Compiler optimizations (incl. reorderings, eliminations)
- Efficient h/w mappings (x86-TSO, Power, ARM)

An intermediate memory model [Podkopaev, L, Vafeiadis POPL'18]



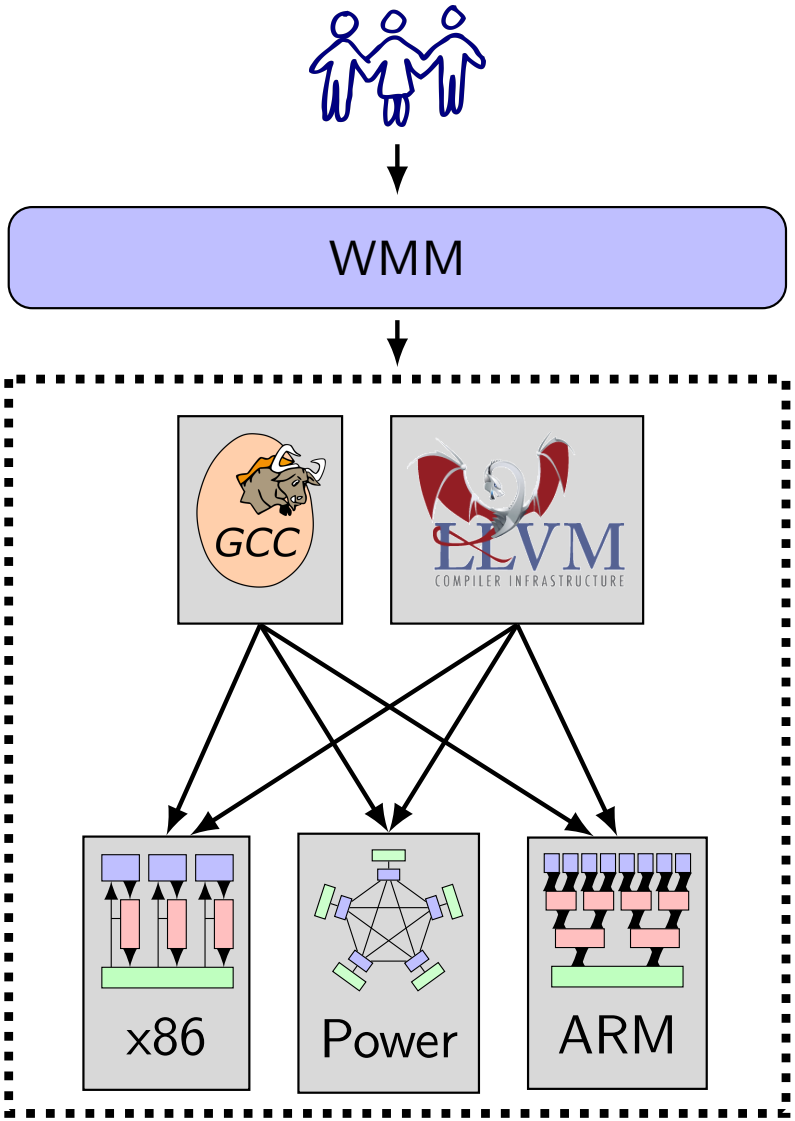
- A *common denominator* of existing models
- Formulated in the *declarative style*
- Simplifies *compilation correctness* proofs

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/