Designing a Programming Language
Shared-Memory Concurrency Semantics

Ori Lahav

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

ceClub -The Technion Computer Engineering Club

February 17, 2021

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?

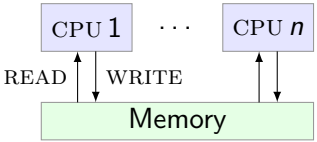
Example: Dekker's mutual exclusion

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):



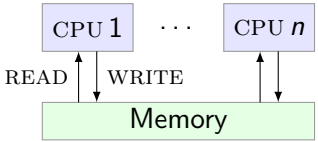
Example: Dekker's mutual exclusion

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).

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;
}
```

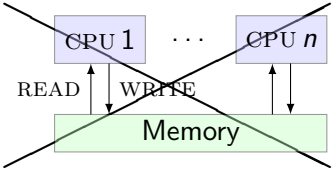
Example: Shared-memory concurrency in C++

```
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



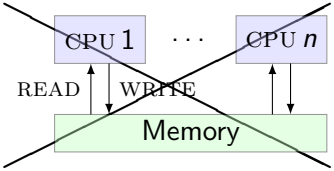
We look for a *substitute for SC*

- What are the possible outcomes of a multi-threaded program in a high-level language?

Typically called a weak memory model (WMM)

- Allows more behaviors than SC.

Weak memory models



We look for a *substitute for SC*

- What are the possible outcomes of a multi-threaded program in a high-level language?

Typically called a weak memory model (WMM)

- Allows more behaviors than SC.

But it is not easy to get right

- The Java memory model (JMM) the the C/C++11 model are both 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

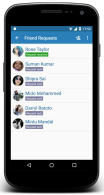
Plan for rest of the talk

1. Challenges for programming language memory models
2. The C/C++11 memory model as a prototype
3. The "out-of-thin-air" problem
4. The "promising semantics" solution

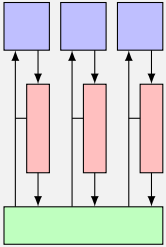
Plan for rest of the talk

1. Challenges for programming language memory models
2. The C/C++11 memory model as a prototype
3. The "out-of-thin-air" problem
4. The "promising semantics" solution

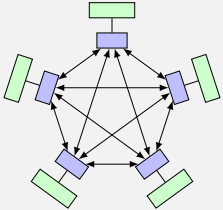
Challenge 1: Various target models



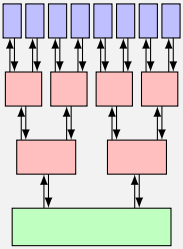
x86-TSO (intel) AMD
(2010)



POWER IBM
(2011)



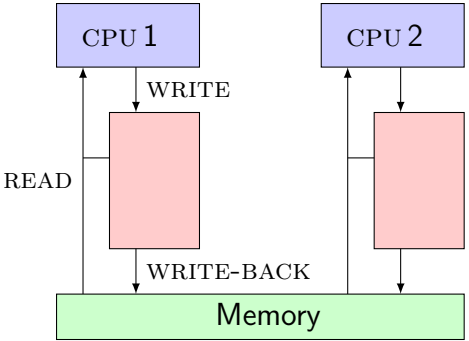
ARMv8 ARM
(2016)



Store buffering in x86-TSO

Initially, $x = y = 0$.

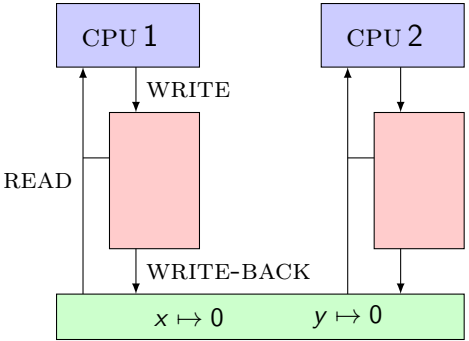
$x := 1;$	$\parallel$	$y := 1;$
$a := y;$ // 0	$\parallel$	$b := x;$ // 0



Store buffering in x86-TSO

Initially, $x = y = 0$.

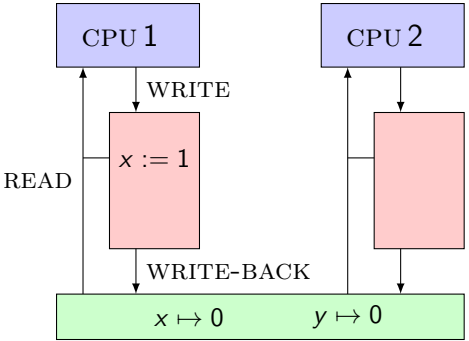
▶ $x := 1;$	▶ $y := 1;$
$a := y;$ // 0	$b := x;$ // 0



Store buffering in x86-TSO

Initially, $x = y = 0$.

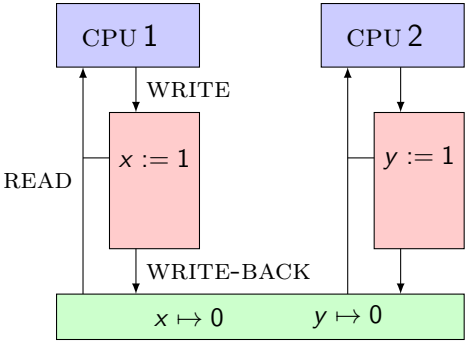
$x := 1;$	$\parallel$	▶ $y := 1;$
▶ $a := y;$ // 0	$\parallel$	$b := x;$ // 0



Store buffering in x86-TSO

Initially, $x = y = 0$.

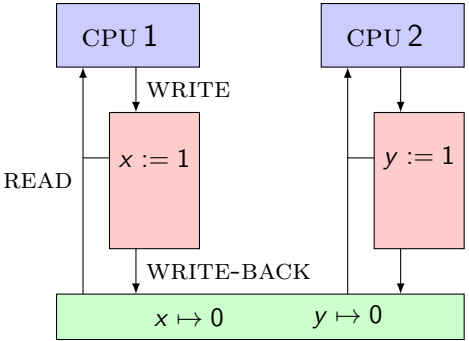
$x := 1;$	$\parallel$	$y := 1;$
$\blacktriangleright\ a := y;$ // 0	$\parallel$	$\blacktriangleright\ b := x;$ // 0



Store buffering in x86-TSO

Initially, $x = y = 0$.

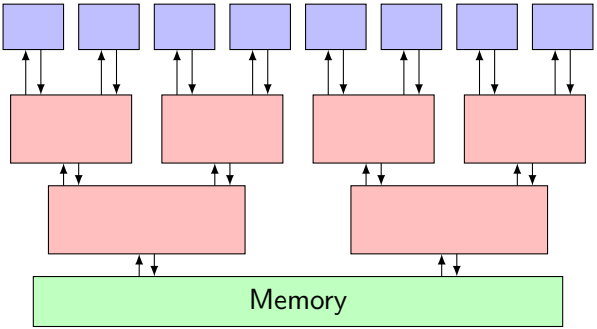
$x := 1;$	$\parallel$	$y := 1;$
fence;		**fence**;
$a := y;$ // 0		$b := x;$ // 0



Load buffering in ARM

Initially, $x = y = 0$.

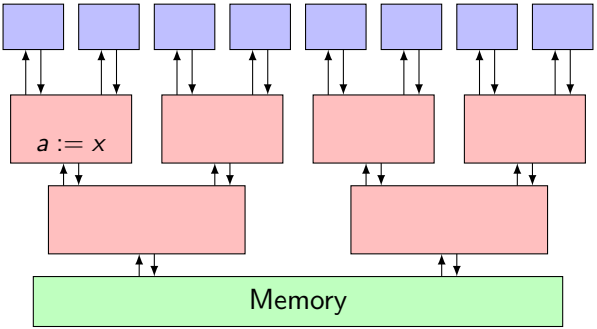
$$\begin{array}{l} a := x; \ \textit{// 1} \\ y := 1; \end{array} \quad \Big\| \quad \begin{array}{l} b := y; \ \textit{// 1} \\ x := b; \end{array}$$



Load buffering in ARM

Initially, $x = y = 0$.

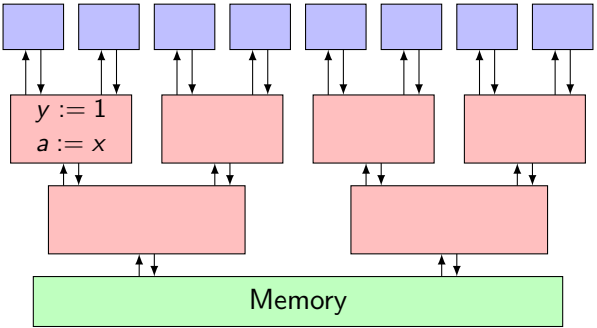
$$
\begin{array}{l}
a := x; \ \ // \ 1 \\
y := 1;
\end{array}
\quad \Big\| \quad
\begin{array}{l}
b := y; \ \ // \ 1 \\
x := b;
\end{array}
$$



Load buffering in ARM

Initially, $x = y = 0$.

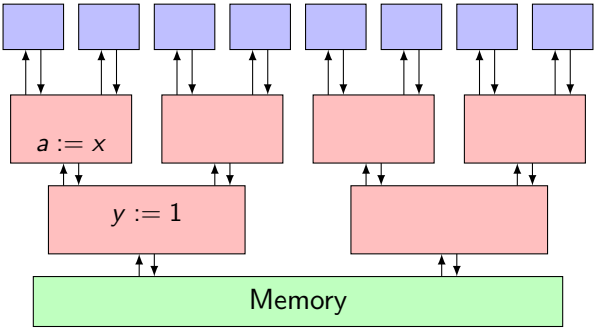
$$
\begin{array}{l||l}
a := x; \ // \ 1 & b := y; \ // \ 1 \\
y := 1; & x := b;
\end{array}
$$



Load buffering in ARM

Initially, $x = y = 0$.

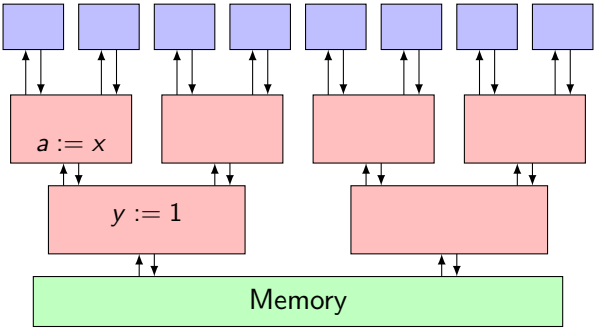
$$
\begin{array}{l}
a := x; \ \ /\!/\, 1 \\
y := 1;
\end{array}
\quad \Big\| \quad
\begin{array}{l}
b := y; \ \ /\!/\, 1 \\
x := b;
\end{array}
$$



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, $x = y = 0$.

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

✗ forbidden under SC

Challenge 2: Compilers stir the pot

Initially, $x = y = 0$.

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

✗ forbidden under SC ✓ allowed under SC

Common sub-expression elimination is unsound under SC

Challenge 3: Transformations do not suffice

Program transformations fail short to explain some weak behaviors.

- In C/C++, no reordering is allowed in the following program:

Message passing (MP)

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

Challenge 3: Transformations do not suffice

Program transformations fail short to explain some weak behaviors.

- In C/C++, no reordering is allowed in the following program:

Message passing (MP)

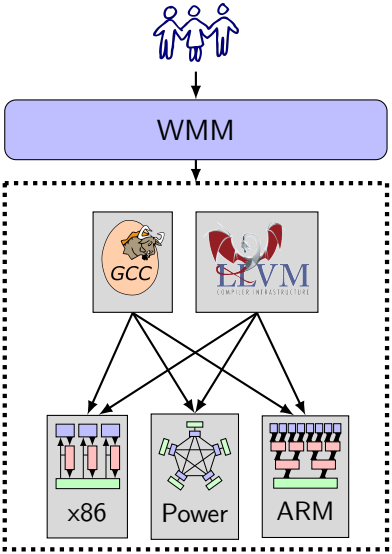
$$\begin{array}{l||l} x := 1; & a := y_{\texttt{acq}};\ \#\ 1 \\ y :=_{\texttt{rel}} 1; & b := x;\ \#\ 0 \end{array}$$

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

Independent reads of independent writes (IRIW)

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

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**

DRF-SC: A fundamental programmability guarantee

DRF-SC guarantee

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

In most cases, programmers can avoid data races by using provided *synchronization mechanisms* (e.g., locks), and need not understand the full semantics.

DRF-SC: A fundamental programmability guarantee

DRF-SC guarantee

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

In most cases, programmers can avoid data races by using provided *synchronization mechanisms* (e.g., locks), and need not understand the full semantics.

Establishing more refined programmability guarantees is an active research area:

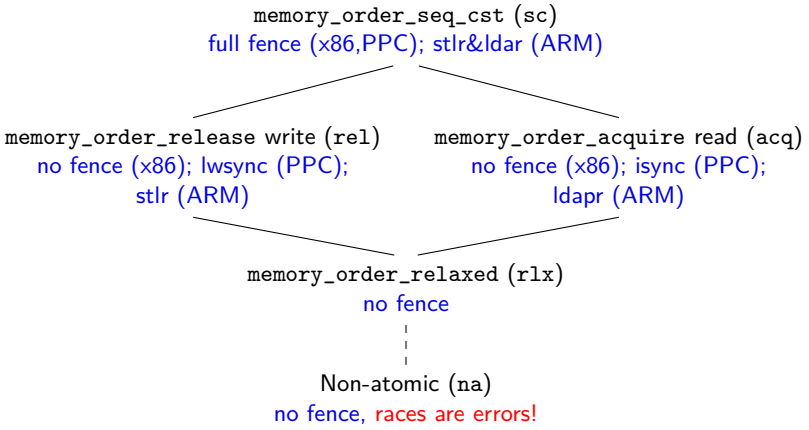
- *Local DRF* for OCaml memory model [Dolan, Sivaramakrishnan, Madhavapeddy PLDI'18]
- DRF wrt fragments *weaker than SC* [Kang, Hur, L, Vafeiadis, Dreyer POPL'17]

The C11 memory model

- Introduced by the ISO C/C++ 2011 standards.
- Serves as a solid basis for:
  - LLVM
  - WebAssembly memory model [Watt et al. OOPSLA 19]
  - JavaScript memory model [Watt et al. PLDI 20]
  - Java 9 [Bender & Palsberg OOPSLA 19]
  - Rust

A spectrum of access modes

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



+ Explicit primitives for language level fences

C11: a declarative memory model

Declarative semantics abstracts away from implementation details.

- Became the "standard" in weak memory models
- Mature formalisms and tools (*e.g.*, Herd [Alglave, Maranget, Tautschnig. TOPLAS'14])

1. a program $\leadsto$ a set of directed graphs.

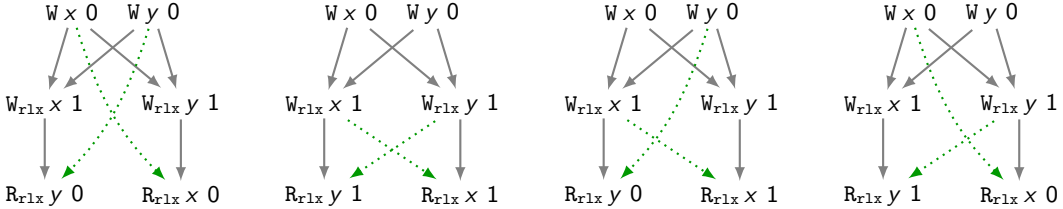
2. The model defines what graphs are *consistent*.

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

[Batty et al. POPL'11]

$$[\![\neg]\!] \;:\; \mathsf{CExp} \to \mathbb{P}(\langle res : \mathsf{Val} \cup \{\bot\}, \mathcal{A} : \mathbb{P}(\mathsf{AName}), \mathsf{lab} : \mathcal{A} \to \mathsf{Act}, \mathsf{sb} : \mathbb{P}(\mathcal{A} \times \mathcal{A}), fst : \mathcal{A}, lst : \mathcal{A}\rangle)$$

$$[\![v]\!] \stackrel{def}{=} \{\langle v, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{ANname} \wedge \mathsf{lab}(a) = \mathsf{skip}\}$$

$$[\![\mathbf{alloc}()]\!] \stackrel{def}{=} \{\langle \ell, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{ANname} \wedge \ell \in \mathsf{Loc} \wedge \mathsf{lab}(a) = \mathrm{A}(\ell)\}$$

$$[\![[v]_X := v']\!] \stackrel{def}{=} \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{ANname} \wedge \mathsf{lab}(a) = \mathrm{W}_X(v, v')\}$$

$$[\![[v]_X]\!] \stackrel{def}{=} \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{ANname} \wedge v' \in \mathsf{Val} \wedge \mathsf{lab}(a) = \mathrm{R}_X(v, v')\}$$

$$[\![\mathbf{CAS}_{X,Y}(v, v_o, v_n)]\!] \stackrel{def}{=} \{\langle v', \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{ANname} \wedge v' \in \mathsf{Val} \wedge v' \neq v_o \wedge \mathsf{lab}(a) = \mathrm{R}_Y(v, v')\}$$
$$\cup\ \{\langle v_o, \{a\}, \mathsf{lab}, \emptyset, a, a\rangle \mid a \in \mathsf{ANname} \wedge \mathsf{lab}(a) \sim \mathrm{RMW}_X(v, v_o, v_n)\}$$

$$[\![\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2]\!] \stackrel{def}{=} \{\langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \mid \langle \bot, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \in [\![E_1]\!]\}$$
$$\cup\ \{\langle res_2, \mathcal{A}_1 \uplus \mathcal{A}_2, \mathsf{lab}_1 \cup \mathsf{lab}_2, \mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(lst_1, fst_2)\}, fst_1, lst_2\rangle$$
$$\langle v_1, \mathcal{A}_1, \mathsf{lab}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \in [\![E_1]\!] \wedge \langle res_2, \mathcal{A}_2, \mathsf{lab}_2, \mathsf{sb}_2, fst_2, lst_2\rangle \in [\![E_2[v_1/x]]\!]\}$$

$$[\![\mathbf{repeat}\ E\ \mathbf{end}]\!] \stackrel{def}{=} \{\langle res_N, \biguplus_{i\in[1..N]} \mathcal{A}_i, \bigcup_{i\in[1..N]} \mathsf{lab}_i, \bigcup_{i\in[1..N]} \mathsf{sb}_i \cup \{(lst_1, fst_2), \ldots, (lst_{N-1}, fst_N)\}, fst_1, lst_N\rangle \mid$$
$$\forall i.\ \langle res_i, \mathcal{A}_i, \mathsf{lab}_i, \mathsf{sb}_i, fst_i, lst_i\rangle \in [\![E]\!] \wedge (i \neq N \implies res_i = 0) \wedge res_N \neq 0\}$$

$$[\![E_1][\!] E_2]\!] \stackrel{def}{=} \{\mathsf{combine}(res_1, res_2), \mathcal{A}_1 \uplus \mathcal{A}_2 \uplus \{a_{\mathrm{fork}}, a_{\mathrm{join}}\}, \mathsf{lab}_1 \cup \mathsf{lab}_2 \cup \{a_{\mathrm{fork}} \mapsto \mathsf{skip}, a_{\mathrm{join}} \mapsto \mathsf{skip}\},$$
$$\mathsf{sb}_1 \cup \mathsf{sb}_2 \cup \{(a_{\mathrm{fork}}, fst_1), (a_{\mathrm{fork}}, fst_2), (lst_1, a_{\mathrm{join}}), (lst_2, a_{\mathrm{join}})\}, a_{\mathrm{fork}}, a_{\mathrm{join}}\rangle \mid$$
$$\langle res_1, \mathcal{A}_1, \mathsf{sb}_1, fst_1, lst_1\rangle \in [\![E_1]\!] \wedge \langle res_2, \mathcal{A}_2, \mathsf{sb}_2, fst_2, lst_2\rangle \in [\![E_2]\!] \wedge a_{\mathrm{fork}}, a_{\mathrm{join}} \in \mathsf{ANname}\}$$

Figure 2. Semantics of closed program expressions.

$$\nexists x.\ \mathsf{hb}(x, x) \qquad \text{(IrreflexiveHB)}$$

$$\forall \ell.\ \ \mathsf{totalorder}(\{a \in \mathcal{A} \mid \mathsf{iswrite}_\ell(a)\}, \mathsf{mo}) \wedge \mathsf{hb}_\ell \subseteq \mathsf{mo} \qquad \text{(ConsistentMO)}$$

$$\mathsf{totalorder}(\{a \in \mathcal{A} \mid \mathsf{isSeqCst}(a)\}, \mathsf{sc}) \wedge \mathsf{hb}_{\mathsf{SeqCst}} \subseteq \mathsf{sc} \wedge \mathsf{mo}_{\mathsf{SeqCst}} \subseteq \mathsf{sc} \qquad \text{(ConsistentSC)}$$

$$\forall b.\ \ \mathsf{rf}(b) \neq \bot \iff \exists \ell, a.\ \mathsf{iswrite}_\ell(a) \wedge \mathsf{isread}_\ell(b) \wedge \mathsf{hb}(a, b) \qquad \text{(ConsistentRFdom)}$$

$$\forall a, b.\ \ \mathsf{rf}(b) = a \implies \exists \ell, v.\ \mathsf{iswrite}_{\ell,v}(a) \wedge \mathsf{isread}_{\ell,v}(b) \wedge \neg \mathsf{hb}(b, a) \qquad \text{(ConsistentRF)}$$

$$\forall a, b.\ \ \mathsf{rf}(b) = a \wedge (\mathsf{mode}(a) = \mathrm{na} \vee \mathsf{mode}(b) = \mathrm{na}) \implies \mathsf{hb}(a, b) \qquad \text{(ConsistentRFna)}$$

$$\forall a, b.\ \ \mathsf{rf}(b) = a \wedge \mathsf{isSeqCst}(b) \implies \mathsf{isc}(a, b) \vee \neg \mathsf{isSeqCst}(a) \wedge (\forall x.\ \mathsf{isc}(x, b) \Rightarrow \neg \mathsf{hb}(a, x)) \qquad \text{(RestrSCReads)}$$

$$\nexists a, b.\ \ \mathsf{hb}(a, b) \wedge \mathsf{mo}(\mathsf{rf}(b), \mathsf{rf}(a)) \wedge \mathsf{locs}(a) \sim \mathsf{locs}(b) \qquad \text{(CoherentRR)}$$

$$\nexists a, b.\ \ \mathsf{hb}(a, b) \wedge \mathsf{mo}(\mathsf{rf}(b), a) \wedge \mathsf{iswrite}(a) \wedge \mathsf{locs}(a) \sim \mathsf{locs}(b) \qquad \text{(CoherentWR)}$$

$$\nexists a, b.\ \ \mathsf{hb}(a, b) \wedge \mathsf{mo}(b, \mathsf{rf}(a)) \wedge \mathsf{iswrite}(b) \wedge \mathsf{locs}(a) \sim \mathsf{locs}(b) \qquad \text{(CoherentRW)}$$

$$\forall a.\ \ \mathsf{isrmw}(a) \wedge \mathsf{rf}(a) \neq \bot \implies \mathsf{mo}(\mathsf{rf}(a), a) \wedge \nexists c.\ \mathsf{mo}(\mathsf{rf}(a), c) \wedge \mathsf{mo}(c, a) \qquad \text{(AtomicRMW)}$$

$$\forall a, b, \ell.\ \ \mathsf{lab}(a) = \mathsf{lab}(b) = \mathrm{A}(\ell) \implies a = b \qquad \text{(ConsistentAlloc)}$$

where $\mathsf{iswrite}_{\ell,v}(a) \stackrel{def}{=} \exists X, v_{\mathrm{old}}.\ \mathsf{lab}(a) \in \{\mathrm{W}_X(\ell, v), \mathrm{RMW}_X(\ell, v_{\mathrm{old}}, v)\}$ $\quad \mathsf{iswrite}_\ell(a) \stackrel{def}{=} \exists v.\ \mathsf{iswrite}_{\ell,v}(a)$

$\mathsf{isread}_{\ell,v}(a) \stackrel{def}{=} \exists X, v_{\mathrm{new}}.\ \mathsf{lab}(a) \in \{\mathrm{R}_X(\ell, v), \mathrm{RMW}_X(\ell, v, v_{\mathrm{new}})\}$ $\quad$ etc.

$\mathsf{rsElem}(a, b) \stackrel{def}{=} \mathsf{sameThread}(a, b) \vee \mathsf{isrmw}(b)$

$$\mathsf{rseq}(a) \stackrel{def}{=} \{a\} \cup \{b \mid \mathsf{rsElem}(a, b) \wedge \mathsf{mo}(a, b) \wedge (\forall c.\ \mathsf{mo}(a, c) \wedge \mathsf{mo}(c, b) \Rightarrow \mathsf{rsElem}(a, c))\}$$

$$\mathsf{sw} \stackrel{def}{=} \{\langle a, b\rangle \mid \mathsf{mode}(a) \in \{\mathrm{rel}, \mathrm{rel\_acq}, \mathrm{sc}\} \wedge \mathsf{mode}(b) \in \{\mathrm{acq}, \mathrm{rel\_acq}, \mathrm{sc}\} \wedge \mathsf{rf}(b) \in \mathsf{rseq}(a)\}$$

$$\mathsf{hb} \stackrel{def}{=} (\mathsf{sb} \cup \mathsf{sw})^+$$

$$\mathsf{hb}_\ell \stackrel{def}{=} \{(a, b) \in \mathsf{hb} \mid \mathsf{iswrite}_\ell(a) \wedge \mathsf{iswrite}_\ell(b)\}$$

$$X_{\mathsf{SeqCst}} \stackrel{def}{=} \{(a, b) \in X \mid \mathsf{isSeqCst}(a) \wedge \mathsf{isSeqCst}(b)\}$$

$$\mathsf{isc}(a, b) \stackrel{def}{=} \mathsf{iswrite}_{\mathsf{locs}(b)}(a) \wedge \mathsf{sc}(a, b) \wedge \nexists c.\ \mathsf{sc}(a, c) \wedge \mathsf{sc}(c, b) \wedge \mathsf{iswrite}_{\mathsf{locs}(b)}(c)$$

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

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

Require the existence of several orders that satisfy certain constraints:

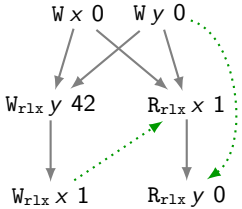
- SC-per-location (a.k.a. coherence)
- Release/acquire synchronization
- Global conditions on SC accesses

excerpt from [Vafeiadis & Narayan OOPSLA'13]

Example: flag-based synchronization

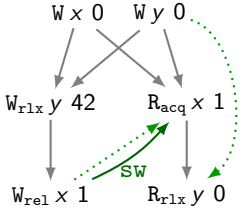
Message passing (MP)

$y :=_{\texttt{rlx}} 42; \;\|\; a := x_{\texttt{rlx}};$ // 1
$x :=_{\texttt{rlx}} 1; \;\|\; b := y_{\texttt{rlx}};$ // 0



Message passing (MP)

$y :=_{\texttt{rlx}} 42; \;\|\; a := x_{\texttt{acq}};$ // 1
$x :=_{\texttt{rel}} 1; \;\|\; b := y_{\texttt{rlx}};$ // 0



Repairing Sequential Consistency in C11 [L, Vafeiadis, Kang, Hur, Dreyer. PLDI'17]

- The semantics of SC accesses is the *most complicated* part of the model.

Repairing Sequential Consistency in C11 [L, Vafeiadis, Kang, Hur, Dreyer. PLDI'17]

- The semantics of SC accesses 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.

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

Repairing Sequential Consistency in C11 [L, Vafeiadis, Kang, Hur, Dreyer. PLDI'17]

- The semantics of SC accesses 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.

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

- The standard committee fixed the specification to solve these problems in C++20.

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` ⊏ `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!**

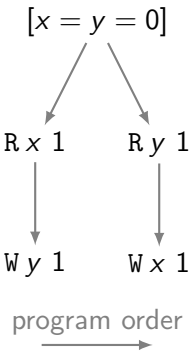
C/C++11 is too weak

non-atomic $\sqsubset$ 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!**



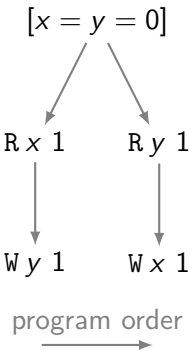
C/C++11 is too weak

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!**



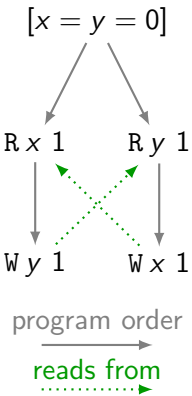
C/C++11 is too weak

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!**



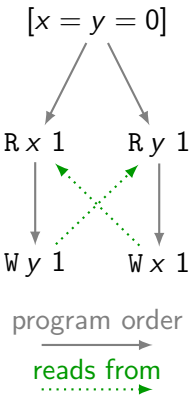
C/C++11 is too weak

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!**



C/C++11 is too weak

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

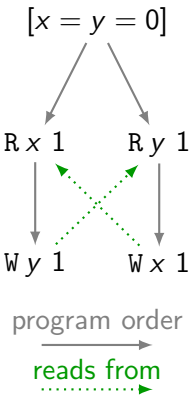
Load-buffering

$a := x;$ // 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 is too weak

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

Load-buffering

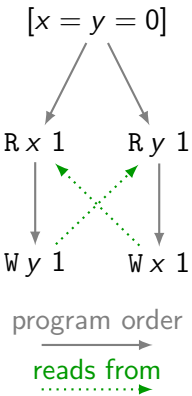
$a := x;$ // 1	$b := y;$ // 1
$y := 1;$	$x := b;$

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



C/C++11 is too weak

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

Load-buffering

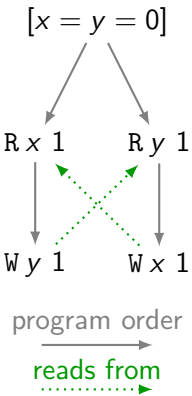
$a := x;$ // 1	$b := y;$ // 1
$y := 1;$	$x := b;$

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)

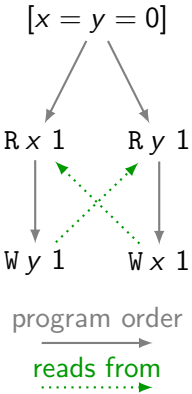


C/C++11 is too weak

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

Load-buffering + control dependency

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



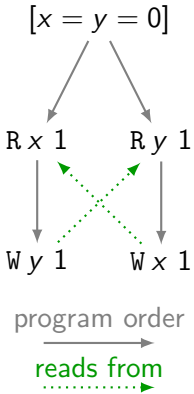
C/C++11 is too weak

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



C/C++11 is too weak

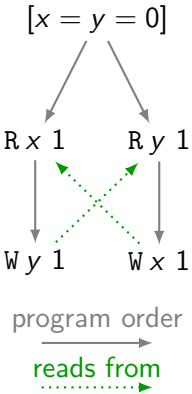
non-atomic $\sqsubset$ relaxed $\sqsubset$ release/acquire $\sqsubset$ sc

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

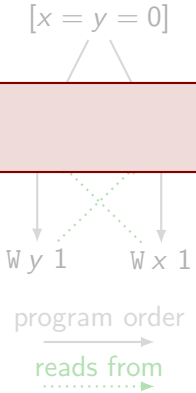
Load-buffering + control dependency

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

The three examples have the same execution graph!

C/C++11 allows this behavior.

The DRF guarantee is broken!



The hardware solution

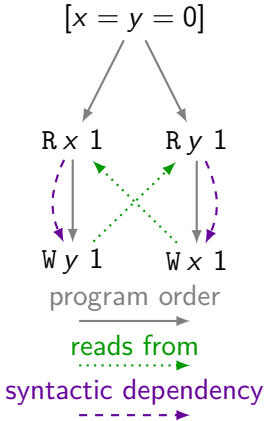
Keep track of syntactic dependencies and forbid dependency cycles.

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

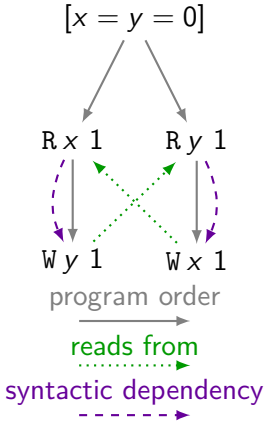
$$a := x;\ /\!/\ 1 \quad \| \quad b := y;\ /\!/\ 1$$
$$y := 1; \quad \| \quad x := b;$$

Load-buffering + data dependency

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

Load-buffering + fake dependency

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



The hardware solution

Keep track of syntactic dependencies and forbid dependency cycles.

Load-buffering

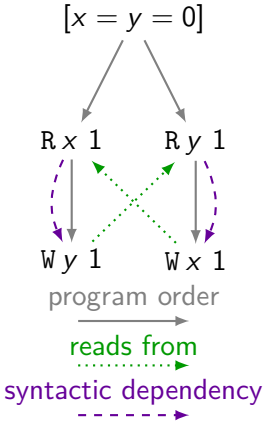
$$\begin{array}{l|l} a := x;\ /\!/\, 1 & b := y;\ /\!/\, 1 \\ y := 1; & x := b; \end{array}$$

Load-buffering + data dependency

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

Load-buffering + fake dependency

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



Unsuitable for PL: **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.”*

Solution

A straightforward solution

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

  [Ou & Demsky. Towards understanding the costs of avoiding out-of-thin-air results. OOPSLA'18]
  Slowdown on ARMv8 is 3.1% on average and 17.6% max (on some benchmarks...)

Solution

A straightforward solution

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

  [Ou & Demsky. Towards understanding the costs of avoiding out-of-thin-air results. OOPSLA'18]
  Slowdown on ARMv8 is 3.1% on average and 17.6% max (on some benchmarks...)

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

- (Modified) compilation schemes are correct.
- DRF holds and no OOTA-values.
- Model checking tools [Kokologiannakis, L, Sagonas, Vafeiadis. POPL'18]
  `http://plv.mpi-sws.org/rcmc/`

Solution

A straightforward solution

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

  [Ou & Demsky. Towards understanding the costs of avoiding out-of-thin-air results. OOPSLA'18]
  Slowdown on ARMv8 is 3.1% on average and 17.6% max (on some benchmarks...)

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

- (Modified) compilation schemes are correct.
- DRF holds and no OOTA-values.
- Model checking tools [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]

A ‘promising’ solution to OOTA

[Kang, Hur, L, Vafeiadis, Dreyer. POPL’17]

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

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

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

$x = y = 0$

$x = 1;$	▶ $y = 1;$
▶ $a = y;$ // 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~~ 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

Coherence Test

$$x = 0$$

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

Simple operational semantics for C11's relaxed accesses

Store-buffering

$$x = y = 0$$

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

Memory

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

T_1's view

x	y
~~0~~ 5	0

T_2's view

x	y
0	~~0~~ 5

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~~ 5	0

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~~ 5	0

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~~ 5	0

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

$a := x;$ // 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 $y := 1$;	▶ $x := y$;

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	
▶ $y := 1;$	$x := y;$
	▶

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
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

$a := x;$ // 1 ‖ $x := y;$
$y := 1;$

Load-buffering + dependency

$a := x;$ // 1 ‖ $x := y;$
$y := a;$

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!

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: read-from-untaken-branch (due to Hans Boehm)

Is this behavior possible?

$a := x;$ // 42
$y := a;$
	$b := y;$
	if $(b = 42)$
	  $c := 1;$
	else
	  $c := 2;$
	  $b := 42;$
	$x := b;$
	print $(c);$ // prints 1

Quiz: read-from-untaken-branch (due to Hans Boehm)

Is this behavior possible?

$a := x;$ // 42
$y := a;$
	$b := y;$
	if $(b = 42)$
	  $c := 1;$
	else
	  $c := 2;$
	  $b := 42;$
	$x := b;$
	print $(c);$ // prints 1

Yes. And it can obtained by standard compiler optimizations!

The full model

We have extended this basic idea to handle:

- Atomic Read-Modify-Writes (e.g., CAS, fetch-and-add)
- Release/acquire accesses and fences
- 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

We have extended this basic idea to handle:

- Atomic Read-Modify-Writes (e.g., CAS, fetch-and-add)
- Release/acquire accesses and fences
- 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)

The full model

We have extended this basic idea to handle:

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

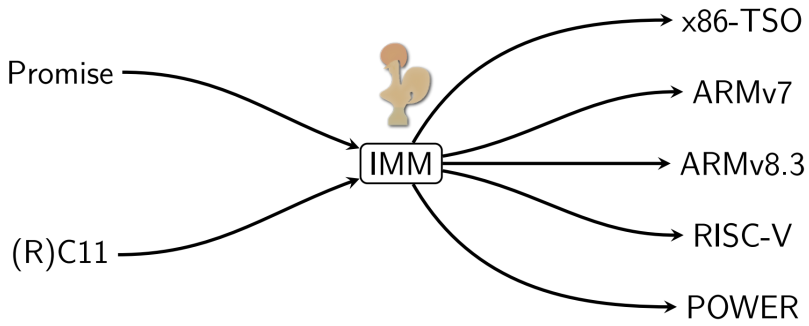
The **Coq** proof assistant

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'19]



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

Certification from current memory is not enough!

$a := \textbf{FADD}(x, 1, \texttt{acq-rel})$ // 0	$b := \textbf{FADD}(x, 1, \texttt{acq-rel})$ // 0
if $a = 0$ **then**	**if** $b = 0$ **then**
$y := 1$	$c := y$ // 1
	if $c = 1$ **then**
	$x := 0$

- The only race is on an *acquire-release RMW*.
- The DRF-RA guarantee entails the annotated behavior should be disallowed.
- Thus, the behavior must be forbidden by the promising model.
- We forbid it by requiring a certification from *any extension* of the current memory.

Complex compilation issues... (1/2)

$$\begin{array}{l} a := y \;/\!/\, 1 \\ z := a \end{array} \;\Big\|\; \begin{array}{l} b := z \;/\!/\, 1 \\ c := \mathbf{FADD}(x, 1) \;/\!/\, 0 \\ y := c + 1 \end{array}$$

- The promising model forbids this behavior.
- But, it is allowed when compiling to ARMv8.
- Register promotion is unsound.

Complex compilation issues... (2/2)

$a := \mathbf{CAS}(x, 0, 1)$ // 1	$x := 42$
if $a < 10$ **then**	$b := y$ // 1
$y := 1$	$x := b$

- The promising model forbids this behavior.
- But, it can be obtained by local compiler optimization + *global value-range analysis*.

Complex compilation issues... (2/2)

$a := \mathbf{CAS}(x, 0, 1)$ // 1	$x := 42$
if $a < 10$ **then**	$b := y$ // 1
$y := 1$	$x := b$

- The promising model forbids this behavior.
- But, it can be obtained by local compiler optimization + *global value-range analysis*.

Promising 2.0 [Lee, Cho, Podkopaev, Chakraborty, Hur, L, Vafeiadis PLDI'20]

These issues were fixed by a better "forall future memory" certification requirement.

An ongoing challenge: A local DRF guarantee

Existing programmability guarantees are non-modular!

$a := \textbf{pop}(S)$	$b := \textbf{pop}(S)$
lock()	**lock**()
process a accessing x, y	process b accessing x, y
unlock()	**unlock**()

- We want to assume SC semantics for the accesses to x and y.
- The stack implementation may have (benign) *races*.
- The (global) DRF-SC guarantee is inapplicable.

An ongoing challenge: A local DRF guarantee

Existing programmability guarantees are non-modular!

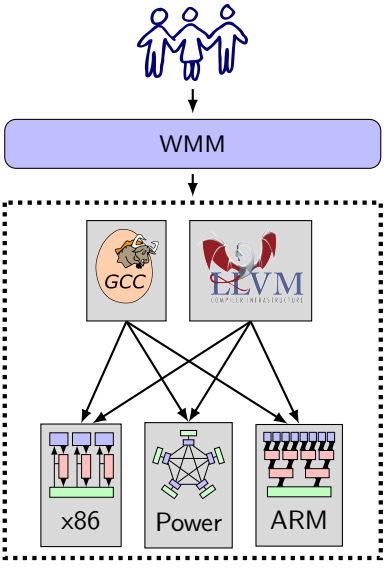
$a := \textbf{pop}(S)$	$b := \textbf{pop}(S)$
lock()	**lock**()
process a accessing x, y	process b accessing x, y
unlock()	**unlock**()

- We want to assume SC semantics for the accesses to x and y.
- The stack implementation may have (benign) *races*.
- The (global) DRF-SC guarantee is inapplicable.

A bad surprise... [Lee, Cho, Hur, L submitted]

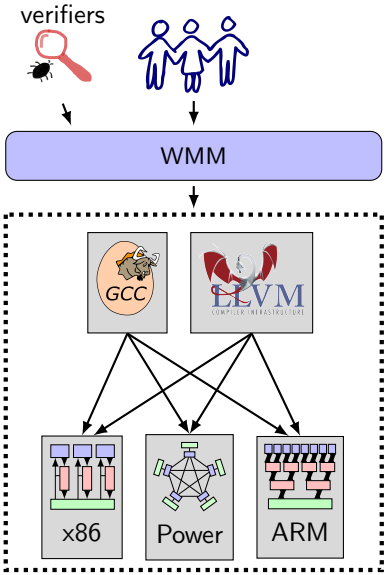
Standard compiler optimizations are inconsistent with local DRF guarantees.

Summary



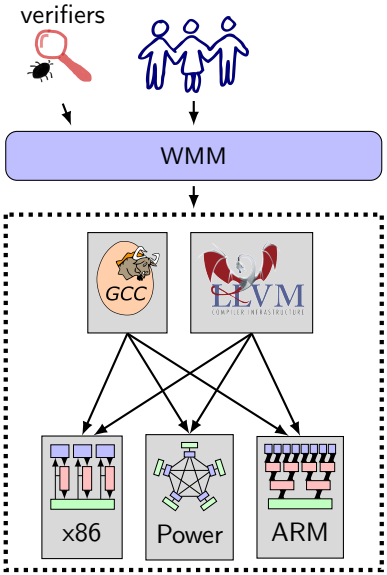
- The challenges in designing a WMM.
- The C/C++11 model.
- C/C++11 is **broken**:
  - Most problems are **locally fixable**.
  - But ruling out **OOTA** requires an entirely different approach.
- The **promising model** may be the solution.

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.

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/