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, \( x = y = 0 \).

\[
\begin{align*}
x & := 1; \\
a & := y; \\
if (a = 0) \text{ then} & \\
\quad /* \text{critical section} */ & \\
y & := 1; \\
b & := x; \\
if (b = 0) \text{ then} & \\
\quad /* \text{critical section} */
\end{align*}
\]

Is it safe? Yes, if we assume sequential consistency (SC):

\text{cpu 1 write read cpu n ... Memory}

No existing hardware implements SC!

▶ SC is very expensive (memory \( \sim \) 100 times slower than CPU).

▶ SC does not scale to many processors.
Example: Dekker’s mutual exclusion

Initially, \( x = y = 0 \).

\[
\begin{align*}
  x &:= 1; & y &:= 1; \\
  a &:= y; & b &:= x; \\
  \textbf{if} \ (a = 0) \ \textbf{then} & & \textbf{if} \ (b = 0) \ \textbf{then} \\
  & /* \textit{critical section} */ & & /* \textit{critical section} */
\end{align*}
\]

Is it safe?

Yes, if we assume sequential consistency (SC):
Example: Dekker’s mutual exclusion

Initially, $x = y = 0$.

\[
\begin{align*}
  x &:= 1; \\
  a &:= y; \quad // 0 \\
  \text{if } (a = 0) \quad \text{then} \\
  &\quad /* \text{critical section} */ \\
  y &:= 1; \\
  b &:= x; \quad // 0 \\
  \text{if } (b = 0) \quad \text{then} \\
  &\quad /* \text{critical section} */
\end{align*}
\]

Is it safe?

Yes, if we assume sequential consistency (SC):

---

Memory

CPU 1 \quad \cdots \quad CPU n

READ \quad WRITE
Example: Dekker’s mutual exclusion

Initially, \( x = y = 0 \).

\[
\begin{align*}
  &x := 1; \\
  &a := y; \quad // 0 \\
  &\text{if } (a = 0) \text{ then} \\
  &\quad /* \text{critical section} */ \\
  &y := 1; \\
  &b := x; \quad // 0 \\
  &\text{if } (b = 0) \text{ then} \\
  &\quad /* \text{critical section} */
\end{align*}
\]

Is it safe?

Yes, if we assume sequential consistency (SC):

No existing hardware implements SC!

- SC is very expensive (memory \( \sim \)100 times slower than CPU).
Example: Shared-memory concurrency in C++

```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++

```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.
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)
Initially, \( x = y = 0 \).

\[ \begin{align*}
x &:= 1; \\
a &:= y; \quad \text{// 0} \\
y &:= 1; \\
b &:= x; \quad \text{// 0}
\end{align*} \]
Initially, $x = y = 0$. 

\[
\begin{align*}
\triangleright & \quad x := 1; \\
& \quad a := y; \quad // 0 \\
\triangleright & \quad y := 1; \\
& \quad b := x; \quad // 0
\end{align*}
\]

![Diagram of store buffering in x86-TSO](image)
Initially, $x = y = 0$. 

\[
\begin{align*}
& \quad x := 1; \\
& \quad y := 1; \\
& \quad a := y; \quad \text{// 0} \\
& \quad b := x; \quad \text{// 0}
\end{align*}
\]
Initially, $x = y = 0$.

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

\begin{align*}
\text{CPU 1} & \quad \text{WRITE} \\
 x := 1 & \\
\text{READ} & \\
\text{WRITE-BACK} & \\
 x \mapsto 0
\end{align*}

\begin{align*}
\text{CPU 2} & \quad \text{WRITE} \\
 y := 1 & \\
\text{READ} & \\
\text{WRITE-BACK} & \\
 y \mapsto 0
\end{align*}
Initially, $x = y = 0$.

\[
x := 1; \quad y := 1; \\
\text{fence}; \quad \text{fence}; \\
a := y; \quad b := x; \quad // 0
\]
Initially, $x = y = 0$.

\[
\begin{align*}
  a &:= x; \quad \text{// 1} \\
  y &:= 1; \\
  b &:= y; \quad \text{// 1} \\
  x &:= b;
\end{align*}
\]
Initially, $x = y = 0$.

\[
a := x; \quad // 1 \\
y := 1; \\
\]

\[
b := y; \quad // 1 \\
x := b; \\
\]
Initially, \( x = y = 0 \).

\[
\begin{align*}
    a & := x; \quad \text{// 1} \quad b := y; \quad \text{// 1} \\
    y & := 1; \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad x := b;
\end{align*}
\]
Initially, $x = y = 0$.

\[
\begin{align*}
a &:= x; \quad // 1 \\
y &:= 1; \\
b &:= y; \quad // 1 \\
x &:= b;
\end{align*}
\]
Initially, $x = y = 0$.

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

b := y;  // 1
x := b;
```
Challenge 2: Compilers stir the pot

Initially, $x = y = 0$.

```plaintext
x := 1;  a := x;
y := 1;  b := y;  // 1
× forbidden under SC
```

```plaintext
forbidden under SC
```
Challenge 2: Compilers stir the pot

Initially, $x = y = 0$.

$\begin{align*}
x &:= 1; \\
y &:= 1; \\
\times\text{ forbidden under SC}
\end{align*}$

$\begin{align*}
a &:= x; \\
b &:= y; \quad \text{// 1} \\
c &:= x; \quad \text{// 0}
\end{align*}$

$\begin{align*}
x &:= 1; \\
y &:= 1; \\
\checkmark\text{ allowed under SC}
\end{align*}$

$\begin{align*}
a &:= x; \\
b &:= y; \quad \text{// 1} \\
c &:= a; \quad \text{// 0}
\end{align*}$

Common sub-expression elimination is \textit{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)

\[
\begin{align*}
  x &:= 1; \\
  y &:=_{\text{rel}} 1; \\
  a &:= y_{\text{acq}}; \quad // 1 \\
  b &:= x; \quad // 0
\end{align*}
\]
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;
  y := rel 1;
  a := y_{acq}; // 1
  b := x; // 0
  ```

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

  ```
  Independent reads of independent writes (IRIW)
  a := x_{acq}; // 1
  b := y_{acq}; // 0
  x := rel 1;
  y := rel 1;
  c := y_{acq}; // 1
  d := x_{acq}; // 0
  ```
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

<table>
<thead>
<tr>
<th>DRF-SC guarantee</th>
</tr>
</thead>
<tbody>
<tr>
<td>no data races under SC $\implies$ only SC behaviors</td>
</tr>
</tbody>
</table>

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**

```
memory_order_seq_cst (sc)
  full fence (x86, PPC); stlr&ldar (ARM)

memory_order_release write (rel)
  no fence (x86); lwsync (PPC);
  stlr (ARM)

memory_order_acquire read (acq)
  no fence (x86); isync (PPC);
  ldapr (ARM)

memory_order_relaxed (rlx)
  no fence

Non-atomic (na)
  no fence, races are errors!
```

+ Explicit primitives for language level fences
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 \( \sim \) a set of directed graphs.

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

Store buffering (SB)

\[ x = y = 0 \]
\[ x := r_{lx} 1; \quad | \quad y := r_{lx} 1; \]
\[ a := y_{rlx}; \quad | \quad b := x_{rlx}; \]

Relations

- Program order, \( po \)
- Reads-from, \( rf \)
C/C++11 formal model

---

CEmp → \(P(x) \lor Val \lor \{\gamma, z\}, A : P(\text{Name})\), lab : A → Act, sb : P(A × A), fast : A, let : A)  
\[\forall (v_1, v_2) \in A \times A \land \gamma(x_1) = \gamma(x_2) \rightarrow v_1 = v_2\]
\[\forall (v_1, v_2) \in A \times A \land \gamma(x_1) \neq \gamma(x_2) \rightarrow v_1 = v_2\]
\[\forall (v_1, v_2) \in A \times A \land \gamma(x_1) = \gamma(x_2) \rightarrow v_1 = v_2\]

[\text{let} \{ (v_1, a), (v_2, b) \} = A : W(\gamma(x_1) \land \gamma(x_2))\]
\[\forall (v_1, v_2) \in A \times A \land \gamma(x_1) = \gamma(x_2) \rightarrow v_1 = v_2\]
\[\forall (v_1, v_2) \in A \times A \land \gamma(x_1) \neq \gamma(x_2) \rightarrow v_1 = v_2\]

\[\forall (v_1, v_2) \in A \times A \land \gamma(x_1) = \gamma(x_2) \rightarrow v_1 = v_2\]

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)**

\[
\begin{align*}
y &:= \text{rlx} 42; & a &:= x_{\text{rlx}}; \quad \mathbin{//} 1 \\
x &:= \text{rlx} 1; & b &:= y_{\text{rlx}}; \quad \mathbin{//} 0
\end{align*}
\]

**Message passing (MP)**

\[
\begin{align*}
y &:= \text{rlx} 42; & a &:= x_{\text{acq}}; \quad \mathbin{//} 1 \\
x &:= \text{rel} 1; & b &:= y_{\text{rlx}}; \quad \mathbin{//} 0
\end{align*}
\]
The semantics of SC accesses is the most complicated part of the model.
The semantics of SC accesses is the most complicated part of the model.

C/C++11 provides too strong semantics (a correctness problem!)

\[
\begin{align*}
a & := x_{\text{acq}}; \quad // 1 \\
b & := y_{\text{sc}}; \quad // 0 \\
x & :=_{\text{sc}} 1; \\
y & :=_{\text{sc}} 1; \\
c & := y_{\text{acq}}; \quad // 1 \\
d & := x_{\text{sc}}; \quad // 0
\end{align*}
\]

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

\[
\begin{align*}
a & := x_{\text{acq}}; \quad // 1 \\
fence_{\text{sc}}; \\
b & := y_{\text{acq}}; \quad // 0 \\
x & :=_{\text{rel}} 1; \\
y & :=_{\text{rel}} 1; \\
c & := y_{\text{acq}}; \quad // 1 \\
fence_{\text{sc}}; \\
d & := x_{\text{acq}}; \quad // 0
\end{align*}
\]

The standard committee fixed the specification to solve these problems in C++20.
The semantics of SC accesses is the most complicated part of the model.

C/C++11 provides too strong semantics (a correctness problem!)

\[
\begin{align*}
a &:= x_{acq}; \quad // 1 \\
b &:= y_{sc}; \quad // 0 \\
x &:=_{sc} 1; \\
y &:=_{sc} 1; \\
c &:= y_{acq}; \quad // 1 \\
d &:= x_{sc}; \quad // 0
\end{align*}
\]

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

\[
\begin{align*}
a &:= x_{acq}; \quad // 1 \\
fence_{sc}; \\
b &:= y_{acq}; \quad // 0 \\
x &:=_{rel} 1; \\
y &:=_{rel} 1; \\
fence_{sc}; \\
c &:= y_{acq}; \quad // 1 \\
d &:= x_{acq}; \quad // 0
\end{align*}
\]

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
- relaxed
- release/acquire
- sc
C/C++11 is too weak

non-atomic  □  relaxed  □  release/acquire  □  sc

Load-buffering

\[
\begin{align*}
a &:= x; \quad \text{// 1} \quad & b &:= y; \quad \text{// 1} \\
y &:= 1; \quad & x &:= b;
\end{align*}
\]

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

\[
\begin{align*}
  a & := x; \quad \text{// 1} & b & := y; \quad \text{// 1} \\
  y & := 1; & x & := b;
\end{align*}
\]

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

\[
\begin{align*}
  & x = y = 0 \\
  & R_x 1 \quad R_y 1 \\
  & W_y 1 \quad W_x 1
\end{align*}
\]

program order
C/C++11 is too weak

non-atomic  □  relaxed  □  release/acquire  □  sc

Load-buffering

\[
\begin{align*}
  a & := x; \quad // 1 \\
  y & := 1; \quad \parallel \quad b & := y; \quad // 1 \\
  x & := b;
\end{align*}
\]

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

\[
[x = y = 0]
\]

\[
\begin{align*}
  & R_x 1 \quad R_y 1 \\
  & \downarrow \quad \downarrow \\
  & W_y 1 \quad W_x 1
\end{align*}
\]

program order
C/C++11 is too weak

non-atomic    □  relaxed  □  release/acquire  □  sc

Load-buffering

\[
\begin{align*}
a & := x; \quad // 1 \\ y & := 1; \\
\end{align*}
\begin{align*}
b & := y; \quad // 1 \\ x & := b;
\end{align*}
\]

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

\[
[x = y = 0]
\]

\[
\begin{array}{c}
R_x 1 \\
W_x 1 \\
\end{array}
\begin{array}{c}
R_y 1 \\
W_y 1 \\
\end{array}
\]

program order
reads from
C/C++11 is too weak

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

\[
\begin{align*}
a & := x; \quad \text{// 1} \\
y & := 1;
\end{align*}
\]

\[
\begin{align*}
b & := y; \quad \text{// 1} \\
x & := b;
\end{align*}
\]

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

Load-buffering + data dependency

\[
\begin{align*}
a & := x; \quad \text{// 1} \\
y & := a;
\end{align*}
\]

\[
\begin{align*}
b & := y; \quad \text{// 1} \\
x & := b;
\end{align*}
\]

Values appear out-of-thin-air! (no hardware/compiler exhibit this behavior)

\[
[x = y = 0]
\]

\[
R_x 1 \quad R_y 1 \quad W_y 1 \quad W_x 1
\]

program order
reads from
C/C++11 is too weak

Load-buffering

\[
\begin{align*}
a &:= x; \quad \text{// 1} & b &:= y; \quad \text{// 1} \\
y &:= 1; & x &:= b;
\end{align*}
\]

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

Load-buffering + data dependency

\[
\begin{align*}
a &:= x; \quad \text{// 1} & b &:= y; \quad \text{// 1} \\
y &:= a; & x &:= b;
\end{align*}
\]

C/C++11 allows this behavior

\[
[x = y = 0]
\]

\[
\begin{align*}
R_x 1 & \quad R_y 1 \\
W_y 1 & \quad W_x 1
\end{align*}
\]

program order
reads from
C/C++11 is too weak

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

Load-buffering

\[ a := x; \quad b := y; \quad y := 1; \quad x := b; \]

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

Load-buffering + data dependency

\[ a := x; \quad b := y; \quad y := a; \quad x := b; \]

C/C++11 allows this behavior **Values appear out-of-thin-air!**

(no hardware/compiler exhibit this behavior)

\[ [x = y = 0] \]

\[
\begin{array}{c}
R_x 1 \\
W_y 1 \\
R_y 1 \\
W_x 1 \\
\end{array}
\]

program order

reads from
C/C++11 is too weak

Load-buffering + control dependency

\[a := x; \quad \text{// 1}\]
\[\text{if } (a = 1) \quad y := 1;\]
\[b := y; \quad \text{// 1}\]
\[\text{if } (b = 1) \quad x := 1;\]

\[x = y = 0\]

\[R_x 1 \quad R_y 1\]
\[W_y 1 \quad W_x 1\]

program order
reads from
C/C++11 is too weak

Load-buffering + control dependency

\[
\begin{align*}
a & := x; \quad // 1 \\
\text{if} \ (a = 1) \\
y & := 1;
\end{align*}
\quad \quad \quad
\begin{align*}
b & := y; \quad // 1 \\
\text{if} \ (b = 1) \\
x & := 1;
\end{align*}
\]

C/C++11 allows this behavior

The DRF guarantee is broken!

\[
\begin{bmatrix}
x = y = 0
\end{bmatrix}
\]

Program order reads from

\[
\begin{align*}
R_x & 1 \\
W_y & 1 \\
R_y & 1 \\
W_x & 1
\end{align*}
\]
C/C++11 is too weak

non-atomic □ relaxed □ release/acquire □ sc

Load-buffering + control dependency

\[
\begin{align*}
a &:= x; \quad \text{// 1} & b &:= y; \quad \text{// 1} \\
\text{if} (a = 1) & & \text{if} (b = 1) \\
y &:= 1; & x &:= 1;
\end{align*}
\]

C/C++11 allows this behavior

The DRF guarantee is broken!
C/C++11 is too weak

The DRF guarantee is broken!

The three examples have the same execution graph!
The hardware solution

Keep track of syntactic dependencies and forbid dependency cycles.

<table>
<thead>
<tr>
<th>Load-buffering</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a := x; // 1 )</td>
</tr>
<tr>
<td>( y := 1; )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load-buffering + data dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a := x; // 1 )</td>
</tr>
<tr>
<td>( y := a; )</td>
</tr>
</tbody>
</table>

\([x = y = 0]\)

R x 1 \[→\] R y 1

W y 1 \[←\] W x 1

program order

reads from

syntactic dependency
The hardware solution

Keep track of syntactic dependencies and forbid dependency cycles.

Load-buffering

\[
\begin{align*}
a &:= x; \quad \text{// 1} \\
y &:= 1;
\end{align*}
\quad \quad \quad
\begin{align*}
b &:= y; \quad \text{// 1} \\
x &:= b;
\end{align*}
\]

Load-buffering + data dependency

\[
\begin{align*}
a &:= x; \quad \text{// 1} \\
y &:= a;
\end{align*}
\quad \quad \quad
\begin{align*}
b &:= y; \quad \text{// 1} \\
x &:= b;
\end{align*}
\]

Load-buffering + fake dependency

\[
\begin{align*}
a &:= x; \quad \text{// 1} \\
y &:= a + 1 - a;
\end{align*}
\quad \quad \quad
\begin{align*}
b &:= y; \quad \text{// 1} \\
x &:= b;
\end{align*}
\]
The hardware solution

Keep track of syntactic dependencies and forbid dependency cycles.

Load-buffering

\[
\begin{align*}
    a & := x; \quad \text{// 1} \\
    y & := 1;
\end{align*}
\]

\[
\begin{align*}
    b & := y; \quad \text{// 1} \\
    x & := b;
\end{align*}
\]

Load-buffering + data dependency

\[
\begin{align*}
    a & := x; \quad \text{// 1} \\
    y & := a;
\end{align*}
\]

\[
\begin{align*}
    b & := y; \quad \text{// 1} \\
    x & := b;
\end{align*}
\]

Load-buffering + fake dependency

\[
\begin{align*}
    a & := x; \quad \text{// 1} \\
    y & := a + 1 - a;
\end{align*}
\]

\[
\begin{align*}
    b & := y; \quad \text{// 1} \\
    x & := b;
\end{align*}
\]

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.”
A straightforward solution

- Disallow $po \cup rf$ 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.

[ 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 straightforward solution

- Disallow $po \cup rf$ 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

- (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/
A straightforward solution

- Disallow po ∪ 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

- (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

\[
\begin{align*}
x &= y = 0 \\
x &= 1; \\
a &= y; \quad \text{// 0} \\
y &= 1; \\
b &= x; \quad \text{// 0}
\end{align*}
\]
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[ x = y = 0 \]

\[ x = 1; \]
\[ a = y; \quad // \ 0 \]

\[ y = 1; \]
\[ b = x; \quad // \ 0 \]

Global memory is a pool of messages of the form

\[ \langle \text{location} : \text{value}@\text{timestamp} \rangle \]

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

Memory

\[ T_1 \text{’s view} \]
\[ x \quad y \]
\[ 0 \quad 0 \]

\[ T_2 \text{’s view} \]
\[ x \quad y \]
\[ 0 \quad 0 \]
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

\[ x = y = 0 \]
\[ x = 1; \]
\[ a = y; \quad // 0 \]
\[ y = 1; \]
\[ b = x; \quad // 0 \]

**Memory**

\[ \langle x : 0@0 \rangle \]
\[ \langle y : 0@0 \rangle \]
\[ \langle x : 1@5 \rangle \]

\[ T_1 \text{’s view} \]
\[
\begin{array}{c|c}
 x & y \\
\hline
 0 & 0 \\
\end{array}
\]

\[ T_2 \text{’s view} \]
\[
\begin{array}{c|c}
 x & y \\
\hline
 0 & 0 \\
\end{array}
\]

- Global memory is a pool of messages of the form
  \[ \langle \text{location} : \text{value}@\text{timestamp} \rangle \]

- Each thread maintains a *thread-local view* recording the last observed timestamp for every location
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[ x = y = 0 \]
\[ x = 1; \]
\[ a = y; \quad // \quad 0 \]
\[ b = x; \quad // \quad 0 \]

Memory
\[
\begin{array}{c}
\langle x : 0@0 \rangle \\
\langle y : 0@0 \rangle \\
\langle x : 1@5 \rangle \\
\langle y : 1@5 \rangle \\
\end{array}
\]

\[ T_1 \text{’s view} \]
\[
\begin{array}{cc}
x & y \\
0 & 0 \\
\end{array}
\]

\[ T_2 \text{’s view} \]
\[
\begin{array}{cc}
x & y \\
0 & 0 \\
\end{array}
\]

- Global memory is a pool of messages of the form

\[ \langle \text{location} : \text{value}@\text{timestamp} \rangle \]

- Each thread maintains a thread-local view recording the last observed timestamp for every location
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[ x = y = 0 \]
\[ x = 1; \]
\[ a = y; \quad // 0 \]
\[ b = x; \quad // 0 \]

Memory

\[ \langle x : 0@0 \rangle \]
\[ \langle y : 0@0 \rangle \]
\[ \langle x : 1@5 \rangle \]
\[ \langle y : 1@5 \rangle \]

T1’s view

\[
\begin{array}{cc}
x & y \\
\hline
0 & 0 \\
5 & 5 \\
\end{array}
\]

T2’s view

\[
\begin{array}{cc}
x & y \\
\hline
0 & x \\
5 & 5 \\
\end{array}
\]

- Global memory is a pool of messages of the form

\[ \langle \text{location} : \text{value}@\text{timestamp} \rangle \]

- Each thread maintains a thread-local view recording the last observed timestamp for every location
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[
\begin{align*}
x &= y = 0 \\
x &= 1; \\
a &= y; \quad // 0 \\
y &= 1; \\
b &= x; \quad // 0
\end{align*}
\]

Memory

\[
\begin{array}{c@{}c@{}c}
\langle x : 0@0 \rangle & \langle y : 0@0 \rangle & \\
\langle x : 1@5 \rangle & \langle y : 1@5 \rangle & \\
\end{array}
\]

\[
\begin{array}{c@{}c}
T_1's \ view & \\
x & y \\
\hline
0 & x \\
5 & 0
\end{array}
\]

\[
\begin{array}{c@{}c}
T_2's \ view & \\
x & y \\
\hline
0 & x \\
5 & 0
\end{array}
\]

- Global memory is a pool of messages of the form
  \[
  \langle \text{location : value}@\text{timestamp} \rangle
  \]

- Each thread maintains a thread-local view recording the last observed timestamp for every location
Simple operational semantics for C11’s relaxed accesses

Store-buffering

\[
\begin{align*}
  x &= y = 0 \\
  x &= 1; \\
  a &= y; \quad // 0 \\
  y &= 1; \\
  b &= x; \quad // 0
\end{align*}
\]

Memory

\[
\begin{array}{c}
  \langle x : 0@0 \rangle \\
  \langle y : 0@0 \rangle \\
  \langle x : 1@5 \rangle \\
  \langle y : 1@5 \rangle \\
\end{array}
\]

Coherence Test

\[
\begin{align*}
  x &= 0 \\
  x := 1; \\
  a &= x; \quad // 2 \\
  x := 2; \\
  b &= x; \quad // 1
\end{align*}
\]

\[
\begin{array}{c}
  \langle x : 0@0 \rangle \\
  \langle y : 0@0 \rangle \\
  \langle x : 1@5 \rangle \\
  \langle y : 1@5 \rangle \\
\end{array}
\]

\[
\begin{array}{c}
  T_1's \ view \\
  x & y \\
  0 & 0 \\
\end{array}
\]

\[
\begin{array}{c}
  T_2's \ view \\
  x & y \\
  5 & 5 \\
\end{array}
\]
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

\[ x = y = 0 \]
\[ x = 1; \]
\[ a = y; \quad \text{∥} 0 \]
\[ y = 1; \]
\[ b = x; \quad \text{∥} 0 \]

**Coherence Test**

\[ x = 0 \]
\[ x := 1; \]
\[ a = x; \quad \text{∥} 2 \]
\[ x := 2; \]
\[ b = x; \quad \text{∥} 1 \]
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

\[
\begin{align*}
    x &= y = 0 \\
    x &= 1; \\
    a &= y; \quad &\text{// 0} \\
    y &= 1; \\
    b &= x; \quad &\text{// 0}
\end{align*}
\]

\[
\begin{array}{c|c|c|c|c|c}
\hline
& T_1 \text{'s view} & & T_2 \text{'s view} \\
\hline
x & y & x & y \\
\hline
\langle x : 0@0 \rangle & \langle y : 0@0 \rangle & \langle x : 1@5 \rangle & \langle y : 1@5 \rangle \\
\hline
0 & 0 & 5 & 5 \\
\hline
\end{array}
\]

**Coherence Test**

\[
\begin{align*}
    x &= 0 \\
    x &:= 1; \\
    a &= x; \quad &\text{// 2} \\
    x &:= 2; \\
    b &= x; \quad &\text{// 1}
\end{align*}
\]

\[
\begin{array}{c|c|c|c|c|c}
\hline
& T_1 \text{'s view} & & T_2 \text{'s view} \\
\hline
x & x & x \\
\hline
\langle x : 0@0 \rangle & \langle x : 1@5 \rangle \\
\hline
0 & 5 & 0 \\
\hline
\end{array}
\]
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

\[
\begin{align*}
x &= y = 0 \\
x &= 1; \\
a &= y; &// 0 \\
b &= x; &// 0 \\
\end{align*}
\]

**Memory**

\[
\begin{array}{c|c}
T_1's \ view & T_2's \ view \\
\hline
\langle x: 0@0 \rangle & \langle x: 0@0 \rangle \\
\langle y: 0@0 \rangle & \langle y: 0@0 \rangle \\
\langle x: 1@5 \rangle & \langle x: 1@5 \rangle \\
\langle y: 1@5 \rangle & \langle y: 1@5 \rangle \\
\end{array}
\]

**Coherence Test**

\[
\begin{align*}
x &= 0 \\
x := 1; \\
a &= x; &// 2 \\
b &= x; &// 1 \\
\end{align*}
\]

**Memory**

\[
\begin{array}{c|c}
T_1's \ view & T_2's \ view \\
\hline
\langle x: 0@0 \rangle & \langle x: 0@0 \rangle \\
\langle x: 1@5 \rangle & \langle x: 1@5 \rangle \\
\langle x: 2@7 \rangle & \langle x: 2@7 \rangle \\
\end{array}
\]
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

\[ x = y = 0 \]

\[ x = 1; \]

\[ a = y; \quad // 0 \]

\[ y = 1; \]

\[ b = x; \quad // 0 \]

**Coherence Test**

\[ x = 0 \]

\[ x := 1; \]

\[ a = x; \quad // 2 \]

\[ x := 2; \]

\[ b = x; \quad // 1 \]
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

\[
\begin{align*}
x &= y = 0 \\
x &= 1; \quad y &= 1; \\
a &= y; \quad // 0 \quad b &= x; \quad // 0
\end{align*}
\]

**Coherence Test**

\[
\begin{align*}
x &= 0 \\
x := 1; \quad x := 2; \\
a &= x; \quad // 2 \quad b &= x; \quad // 1
\end{align*}
\]
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.
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.
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.
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.
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

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.
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

\[ a := x; \quad // 1 \quad x := y; \]
\[ y := 1; \]

Load-buffering + dependency

\[ a := x; \quad // 1 \quad x := y; \]
\[ y := a; \]

Memory

\[ \langle x : 0@0 \rangle \]
\[ \langle y : 0@0 \rangle \]
\[ \langle y : 1@5 \rangle \]
\[ \langle x : 1@5 \rangle \]

\[ T_1 \text{'s view} \]
\[ x \quad y \]
\[ 5 \quad 5 \]

\[ T_2 \text{'s view} \]
\[ x \quad y \]
\[ 5 \quad 5 \]

Must not admit the same execution!
Certified promises

Thread-local certification

A thread can promise to write a message, if it can \textit{thread-locally certify} that its promise will be fulfilled.

Load-buffering

\[
\begin{align*}
    a & := x; \quad // 1 \\
    y & := 1;
\end{align*}
\]

\[
\begin{align*}
    x & := y;
\end{align*}
\]

\textit{T}_1 \text{ may promise } y := 1, \text{ since it is able to write } y := 1 \text{ by itself.}

Load buffering + fake dependency

\[
\begin{align*}
    a & := x; \quad // 1 \\
    y & := a + 1 - a;
\end{align*}
\]

\[
\begin{align*}
    x & := y;
\end{align*}
\]

Load buffering + dependency

\[
\begin{align*}
    a & := x; \quad // 1 \\
    y & := a;
\end{align*}
\]

\[
\begin{align*}
    x & := y;
\end{align*}
\]

\textit{T}_1 \text{ may NOT promise } y := 1, \text{ since it is not able to write } y := 1 \text{ by itself.}
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
```
Is this behavior possible?

```plaintext
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)

Results

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

The Coq proof assistant

VERIFIED
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 Coq proof assistant

VERIFIED
An intermediate memory model

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

[Podkopaev, L, Vafeiadis POPL’19]
Certification from current memory is not enough!

\[ a := \text{FADD}(x, 1, \text{acq-rel}) \parallel 0 \]
\[ \text{if } a = 0 \text{ then } y := 1 \]
\[ b := \text{FADD}(x, 1, \text{acq-rel}) \parallel 0 \]
\[ \text{if } b = 0 \text{ then } c := y \parallel 1 \]
\[ \text{if } c = 1 \text{ then } x := 0 \]

- 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.
The promising model forbids this behavior.

But, it is allowed when compiling to ARMv8.

Register promotion is unsound.
The promising model forbids this behavior.

But, it can be obtained by local compiler optimization + *global value-range analysis*. 
Complex compilation issues… (2/2)

\[ a := \text{CAS}(x, 0, 1) \quad // \quad x := 42 \]
\[ \text{if } a < 10 \text{ then } \quad // \quad b := y \quad // \quad x := b \]
\[ y := 1 \]

- 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!

\[
\begin{align*}
  a & := \text{pop}(S) \\
  \text{lock}() \\
  \text{process } a \text{ accessing } x, y \\
  \text{unlock}() \\
\end{align*}
\quad \quad \quad
\begin{align*}
  b & := \text{pop}(S) \\
  \text{lock}() \\
  \text{process } b \text{ accessing } x, y \\
  \text{unlock}()
\end{align*}
\]

- We want to assume SC semantics for the accesses to \( x \) and \( y \).
- The stack implementation may have (benign) \textit{races}.
- The (global) DRF-SC guarantee is \textit{inapplicable}.
An ongoing challenge: A local DRF guarantee

Existing programmability guarantees are non-modular!

\[ a := \text{pop}(S) \]
\[ \text{lock()} \]
\[ \text{process } a \text{ accessing } x, y \]
\[ \text{unlock()} \]

\[ b := \text{pop}(S) \]
\[ \text{lock()} \]
\[ \text{process } b \text{ accessing } x, y \]
\[ \text{unlock()} \]

- 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.
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.
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.
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/