Weak Memory Concurrency in C/C++11

Ori Lahav

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

Hydra
Distributed Computing Conference

July 11, 2019
Example: Dekker’s mutual exclusion

Initially, $x = y = 0$.

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

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

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

- CPU 1
- write
- read
- CPU n...

Memory

No existing hardware implements SC!

- SC is very expensive (memory ∼ 100 times slower than CPU).
- SC does not scale to many processors.
Example: Dekker’s mutual exclusion

Initially, $x = y = 0$.

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

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

Is it safe?

Yes, if we assume sequential consistency (SC):

1. **CPU 1**: write, read
2. **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, $x = y = 0$.

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

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

Is it safe?

Yes, if we assume sequential consistency (SC):

---

Memory

---

CPU 1  \cdots  CPU n

READ  WRITE
Example: Dekker’s mutual exclusion

Initially, $x = y = 0$.

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

if ($a = 0$) then  
/* critical section */

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

if ($b = 0$) then  
/* critical section */

Is it safe?

Yes, if we assume sequential consistency (SC):

No existing hardware implements SC!

- SC is very expensive (memory $\sim$100 times slower than CPU).
- SC does not scale to many processors.
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;
}
```
```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:

**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.
“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.”
Plan for rest of the talk

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

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

- x86-TSO (2010) by Intel and AMD
- POWER (2011) by IBM
- ARMv8 (2016) by ARM
Initially, $x = y = 0$.

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

- $x := 1$;
- $a := y; // 0$
- $y := 1$;
- $b := x; // 0$
Initially, $x = y = 0$.

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

$x := 1; \quad \|
\quad y := 1; \\
\quad ▶ a := y; \quad // 0 \quad \|
\quad ▶ b := x; \quad // 0$
Initially, \( x = y = 0 \).

\[
\begin{align*}
  x &:= 1; \quad \text{fence;} \\
  a &:= y; \quad // \ 0 \\
  y &:= 1; \quad \text{fence;} \\
  b &:= x; \quad // \ 0
\end{align*}
\]
Load buffering in ARM

Initially, $x = y = 0$.

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

Initially, $x = y = 0$.

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

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

Initially, $x = y = 0$.

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

$$b := y; \ // 1$$
$$x := b;$$
Load buffering in ARM

Initially, $x = y = 0$.

$$a := x; \quad b := y;$$

$$y := 1; \quad x := b;$$
Load buffering in ARM

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

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

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

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

\[
\begin{align*}
  c &:= x; \quad // 0
\end{align*}
\]

\( \times \) forbidden under SC
Challenge 2: Compilers stir the pot

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

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

\( \times \) forbidden under SC

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

\( \checkmark \) 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)

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

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

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

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

Races are errors

Atomic

Welcome to the expert mode
The C11 memory model: Atomics

Two types of accesses

Ordinary (Non-Atomic)

Races are errors

Atomic

Welcome to the expert mode

DRF (data race freedom) guarantee

no data races $\Rightarrow$ only
under SC $\Rightarrow$ 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 $\sim$ 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)

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

Relations

- Program order, po
- Reads-from, rf
\[
\begin{align*}
\text{\text{Figure 2. Semantics of closed program expressions.}} \\
\forall \ell. \text{ totalorder} (\forall a \in A \mid \text{iswrite}_\ell(a), \text{mo}) \land \text{hb}_\ell \subseteq \text{mo} \quad \text{(IrreflexiveHB)}
\end{align*}
\]
\[
\forall \ell. \text{ totalorder} (\forall a \in A \mid \text{isSeqCst}(a)), \text{sc}) \land \text{hb}_\text{SeqCst} \subseteq \text{sc} \land \text{mode}_\text{SeqCst} \subseteq \text{sc} \quad \text{(ConsistentSC)}
\]
\[
\forall a, b. \text{ rf}(b) \neq \perp \implies \exists \ell. a. \text{iswrite}_\ell(a) \land \text{isread}_\ell(b) \land \text{hb}(a, b) \quad \text{(ConsistentRFdom)}
\]
\[
\forall a, b. \text{ rf}(b) = a \implies \exists \ell. v. \text{iswrite}_\ell(v) \land \text{isread}_\ell(b) \land \neg \text{hb}(a, b) \quad \text{(ConsistentRF)}
\]
\[
\forall a, b. \text{ rf}(b) = a \land (\text{mode}(a) = \text{na} \lor \text{mode}(b) = \text{na}) \implies \text{hb}(a, b) \quad \text{(ConsistentRFna)}
\]
\[
\forall a, b. \text{rf}(b) = a \land \text{isSeqCst}(b) \implies \text{isc}(a, b) \lor \neg \text{isSeqCst}(a) \land (\forall \ell. \text{isc}(x, b) \implies \neg \text{hb}(a, x)) \quad \text{(RestrSCReads)}
\]
\[
\forall a, b. \text{hb}(a, b) \land \text{mo}(\text{rf}(b), a, b) \land \text{loc}(a) = \text{loc}(b) \quad \text{(CoherentRR)}
\]
\[
\forall a, b. \text{hb}(a, b) \land \text{mo}(\text{rf}(b), a, b) \land \text{isc}(a) \land \text{loc}(a) = \text{loc}(b) \quad \text{(CoherentWR)}
\]
\[
\forall a, b. \text{hb}(a, b) \land \text{mo}(\text{rf}(b), a, b) \land \text{isc}(b) \land \text{loc}(a) = \text{loc}(b) \quad \text{(CoherentRW)}
\]
\[
\forall a. \text{isrmw}(a) \land \text{rf}(a) \neq \perp \implies \text{mo}(\text{rf}(a), a) \land \neg \text{isc}(a, b) \quad \text{(AtomicRMW)}
\]
where \text{iswrite}_\ell(v) \text{ def } \exists X, v_{\text{old}}, v_{\text{new}}. \text{lab}(a) \in \{X, v_{\text{old}}, v_{\text{new}}\} \quad \text{iswrite}_\ell(a) \text{ def } \exists v. \text{iswrite}_\ell(v, a) \text{etc.}
\]
\[
\text{rsEle}m(a, b) \text{ def } \exists X, v. \text{lab}(a) = \{X, v\} \quad \text{rsEle}m(a, b) \text{ def } \exists X, v. \text{lab}(a) = \{X, v\} \text{ etc.}
\]
\[
\text{rseq}(a) \text{ def } \{a\} \cup \{b\} \quad \text{rsEle}m(a, b) \lor \text{mo}(a, b) \land \forall c. \text{mo}(a, c) \land \text{mo}(c, b) \implies \text{rsEle}m(a, c)
\]
\[
\text{sw} \text{ def } \{a, b\} \land \text{mode}(a) \in \{\text{acq}, \text{rel}_{\text{acq}}, \text{sc}\} \land \text{mode}(b) \in \{\text{acq}, \text{rel}_{\text{acq}}, \text{sc}\} \land \text{rf}(b) \in \text{rseq}(a)
\]
\[
\text{hb} \text{ def } \{\text{sb} \lor \text{sw}\} +
\]
\[
\text{hb}_{\ell} \text{ def } \{a, b\} \in \text{hb} \land \text{iswrite}_\ell(a) \land \text{iswrite}_\ell(b)
\]
\[
\text{X}_{\text{SeqCst}} \text{ def } \{a, b\} \in X \land \text{isSeqCst}(a) \land \text{isSeqCst}(b)
\]
\[
\text{isc}(a, b) \text{ def } \text{iswrite}_\text{loc}(a, b) \land \text{sc}(a, b) \land \neg \text{isc}(c, a) \land \text{sc}(c, b) \land \text{iswrite}_\text{loc}(c)
\]

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

\[
\begin{align*}
c : W(1, 1) \xrightarrow{\text{rf}} a : R(1, 1) \\
\uparrow^{\text{mo}} b \xrightarrow{\text{hb}} c : W(1, 1) \\
\end{align*}
\]
\[
\begin{align*}
d : W(2, 2) \xrightarrow{\text{rf}} b : R(2, 2) \\
\end{align*}
\]
\[
\text{violates CoherentRR}
\]

\[
\begin{align*}
c : W(1, 2) \xrightarrow{\text{mo}} a : W(1, 1) \\
\uparrow^{\text{rf}} b \xrightarrow{\text{hb}} c : W(1, 1) \\
\end{align*}
\]

\[
\begin{align*}
a \xrightarrow{\text{rf}} b \text{ means } a = \text{rf}(b) \\
a \xrightarrow{\text{mo}} b \text{ means } \text{mo}(a, b) \\
a \xrightarrow{\text{hb}} b \text{ means } \text{hb}(a, b)
\end{align*}
\]

\[
\text{Figure 4. Sample executions violating coherency conditions (Batty et al. 2011).}
\]
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 S s.t. the following hold:

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

Namely, the following is **disallowed**:

\[
W \times S \xrightarrow{\text{rf}} W \times S \xrightarrow{\text{R}} W \times S
\]
$x = y = 0$

$x := y_{rlx} \ 1; \ \parallel \ y := y_{rlx} \ 1;

a := y_{rlx}; \ // \ 0 \ \parallel \ b := x_{rlx}; \ // \ 0$

---

program order

reads from

not SC-consistent!
**Definition (SC-per-location)**

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

- $S_x$ is a total order on the events of $G$ that access $x$.
- $(\text{po} \cup \text{rf}); S_x = \emptyset$.
- If $\langle a, b \rangle \in \text{rf}$ then there does not exist $c \in W_x$ such that $\langle a, c \rangle \in S_x$ and $\langle c, b \rangle \in S_x$.

Namely, the following is disallowed:
SC-per-location: Example 1

\[ x = y = 0 \]

\[ x :=_{rlx} 1; \quad y :=_{rlx} 1; \]
\[ a := y_{rlx}; \quad // 0 \quad b := x_{rlx}; \quad // 0 \]

program order

reads from

satisfies SC-per-location!
SC-per-location: Example 2

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

The program order 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)](image)

▶ Also: we cannot implement locks.
Synchronization in C/C++11 through examples

```c
int y = 0;
int x = 0;
y = 42; || if(x == 1){
x = 1; || print(y);
} ||
```
Synchronization in C/C++11 through examples

```c
int y = 0;
int x = 0;
y = 42; if(x == 1){
    x = 1; print(y);
}
```
Synchronization in C/C++11 through examples

1

```cpp
int y = 0;
int x = 0;
y = 42; if(x == 1){
    x = 1; print(y);
} race
```
Synchronization in C/C++11 through examples

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

2
```c
int y = 0;
atomic<int> x = 0;
y = 42;  // if(x_{rlx} == 1){
x =_{rlx} 1;  // race
    print(y);
}
```

Note: The code snippet on the left demonstrates a race condition due to the lack of synchronization, while the right uses C++11's atomic operations to ensure correct synchronization.
Synchronization in C/C++11 through examples

1

```c
int y = 0;
int x = 0;
y = 42;  // if(x == 1){
x = 1,  // print(y);
}  
```

2

```c
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;   // race
   print(y);
   }        // race
```  

2. ```
   int y = 0;
   atomic<int> x = 0;
   y = 42;  // if(x_{rlx} == 1){
   x =_{rlx} 1; // race
   print(y);
   }        // race
```  

3. ```
   int y = 0;
   atomic<int> x = 0;
   y = 42;  // if(x_{acq} == 1){
   x =_{rel} 1; // race
   print(y);
   }        // race
```
Synchronization in C/C++11 through examples

1

```c
int y = 0;
int x = 0;
y = 42;  \text{if}(x == 1)\{
x = 1,  \quad \text{race traverses here} \quad \text{print}(y);
}\}
```

2

```c
atomic<int> x = 0;
y = 42;  \text{if}(x_{rlx} == 1)\{
x =_{rlx} 1;  \quad \text{race traverses here} \quad \text{print}(y);
}\}
```

3

```c
int y = 0;
atomic<int> x = 0;
y = 42;  \text{if}(x_{acq} == 1)\{
x =_{rel} 1;  \quad \text{race traverses here} \quad \text{print}(y);
}\}
```
Synchronization in C/C++11 through examples

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

2
atomic<int> x = 0;
y = 42;  // if(x_rlx == 1) {
x = rlx 1;  // race
print(y);
}

3
int y = 0;
atomic<int> x = 0;
y = 42;  // if(x_acq == 1) {
x = rel 1;  // rf
print(y);
}
Synchronization in C/C++11 through examples

1
```c
int y = 0;
int x = 0;
y = 42; // if(x == 1){
x = 1,  \(\text{race}\) print(y);
}
```

2
```c
int y = 0;
atomic<int> x = 0;
y = 42; // if(x\_rlx == 1){
x =\_rlx 1;  \(\text{race}\) print(y);
}
```

3
```c
int y = 0;
atomic<int> x = 0;
y = 42;  // if(x\_acq == 1){
x =\_rel 1;  \(\text{race}\) print(y);
}
```

4
```c
int y = 0;
atomic<int> x = 0;
y = 42;  // if(x\_rlx == 1){
fence\_rel;  \(\text{race}\) fence\_acq;
x =\_rlx 1;  \(\text{race}\) print(y);
}
```
Synchronization in C/C++11 through examples

1

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

2

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

3

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

4

```c
int y = 0;
atomic<int> x = 0;
y = 42;  
if(x rlx == 1){
fence rel;  
x =rlx 1,  
fence acq;  
    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);
   }
The “synchronizes-with” relation

\[
\begin{align*}
y &= 42; \quad \text{if}(x_{\text{acq}} == 1)\
x &=_{\text{rel}} 1; \quad \text{print}(y);
\end{align*}
\]

\[
\begin{align*}
y &= 42; \quad \text{if}(x_{\text{rlx}} == 1)\
x &=_{\text{rel}} 1; \quad \text{fence}_{\text{acq}}; \quad \text{print}(y);
\end{align*}
\]
The “synchronizes-with” relation: Release sequences

Note: the latter case will be deprecated in C++20.
The “happens-before” relation

Definition (happens-before)

\[
\begin{align*}
\text{po} & \quad a \rightarrow b \\
\text{sw} & \quad a \rightarrow b \\
\text{hb} & \quad a \rightarrow b \quad a \rightarrow b \quad a \rightarrow b
\end{align*}
\]

- \(\text{hb}\) should be acyclic.
- The SC-per-location orders should never contradict \(\text{hb}\).

\[
\begin{align*}
\text{Wh} & \times 0 \quad \text{Wy} \times 0 \\
\text{Wrlx} & \times 1 \quad \text{Racq} \times 1 \\
\text{Wrel} & \times 1 \quad \text{Rrlx} \times 0
\end{align*}
\]
SC accesses and fences

How to guarantee only SC behaviors (i.e., $a = 1 \lor b = 1$)?
SC accesses and fences

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

$$x :=_{\text{sc}} 1; \quad y :=_{\text{sc}} 1; \quad x :=_{\text{rlx}} 1; \quad y :=_{\text{rlx}} 1;$$

$$a := y_{\text{sc}}; \quad b := x_{\text{sc}}; \quad \text{fence}_{\text{sc}}; \quad a := y_{\text{rlx}}; \quad b := x_{\text{rlx}};$$
SC semantics

- The semantics of SC atomics is the *most complicated* part of the model.
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!)

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

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

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

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

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

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

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

- Recently, the standard committee fixed the specification following:

  [Repairing Sequential Consistency in C/C++11. L, Vafeiadis, Kang, Hur, Dreyer. PLDI’17]
The “out-of-thin-air” problem
C/C++11 is too weak

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

non-atomic □ relaxed □ release/acquire □ sc

Load-buffering

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

b &:= y; \quad // 1 \\
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

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

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

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

\[
[x = y = 0]
\]

\[
\begin{align*}
R x & 1 \\
R y & 1 \\
W y & 1 \\
W x & 1
\end{align*}
\]

program order
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!**

[\[x = y = 0]\]

\[
\begin{array}{c}
R x 1 \\
W y 1 \\
R y 1 \\
W x 1
\end{array}
\]

program order
C/C++11 is too weak

non-atomic  □  relaxed  □  release/acquire  □  sc

Load-buffering

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

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

\[
[x = y = 0]
\]

program order
reads from

\[
\begin{align*}
    R x 1 & \quad R y 1 \\
    W y 1 & \quad W x 1
\end{align*}
\]
C/C++11 is too weak

non-atomic  □  relaxed  □  release/acquire  □  sc

Load-buffering

<table>
<thead>
<tr>
<th>(a := x; \quad \text{// 1})</th>
<th>(b := y; \quad \text{// 1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y := 1;)</td>
<td>(x := b;)</td>
</tr>
</tbody>
</table>

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

\[ [x = y = 0] \]

\(R \times 1\) \hspace*{1cm} \(R \times y 1\)

\(W \times y 1\) \hspace*{1cm} \(W \times x 1\)

program order \hspace*{1cm} reads from

\[ x = y = 0 \]
C/C++11 is too weak

non-atomic □ relaxed □ release/acquire □ sc

Load-buffering

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

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

Load-buffering + data dependency

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

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

\[
[x = y = 0]
\]

program order reads from

\[
R x 1 \quad R y 1
\]

\[
W y 1 \quad W x 1
\]
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!**

Load-buffering + data dependency

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

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

C/C++11 allows this behavior

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

\[
\begin{align*}
x &= y &= 0
\end{align*}
\]

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

program order

reads from
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!**

Load-buffering + data dependency

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

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

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

The DRF guarantee is broken!

\([x = y = 0]\)

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

program order

reads from
C/C++11 is too weak

non-atomic □ relaxed □ release/acquire □ sc

Load-buffering + control dependency

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

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

C/C++11 allows this behavior

\[ [x = y = 0] \]

program order reads from

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

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

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

program order

\[
\begin{align*}
W \ y &\rightarrow \ W \ x \\
\text{reads from}
\end{align*}
\]

The three examples have the same execution graph!

The DRF guarantee is broken!
The hardware solution

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

<table>
<thead>
<tr>
<th>Load-buffering</th>
</tr>
</thead>
</table>
| $a := x; \ // 1$ || $b := y; \ // 1$
| $y := 1;$ || $x := b;$

<table>
<thead>
<tr>
<th>Load-buffering + data dependency</th>
</tr>
</thead>
</table>
| $a := x; \ // 1$ || $b := y; \ // 1$
| $y := a;$ || $x := b;$

$[x = y = 0]$

Program order

- Reads from

Syntactic dependency

Compilers do not preserve syntactic dependencies.
The hardware solution

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

### Load-buffering

- \( a := x; \quad // 1 \)
- \( y := 1; \)
- \( b := y; \quad // 1 \)
- \( x := b; \)

### Load-buffering + data dependency

- \( a := x; \quad // 1 \)
- \( y := a; \)
- \( b := y; \quad // 1 \)
- \( x := b; \)

### Load-buffering + fake dependency

- \( a := x; \quad // 1 \)
- \( y := a + 1 - a; \)
- \( b := y; \quad // 1 \)
- \( x := b; \)

\([x = y = 0]\)

\( [R x 1 \quad R y 1] \)

\( [W y 1 \quad W x 1] \)

program order

reads from

syntactic dependency
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; \quad b := y; ]</td>
</tr>
<tr>
<td>[ y := 1; \quad x := b; ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load-buffering + data dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ a := x; \quad b := y; ]</td>
</tr>
<tr>
<td>[ y := a; \quad x := b; ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Load-buffering + fake dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ a := x; \quad b := y; ]</td>
</tr>
<tr>
<td>[ y := a + 1 - a; \quad x := b; ]</td>
</tr>
</tbody>
</table>

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.
A straightforward solution

- Disallow \( \text{po} \cup \text{rf} \) 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]
A straightforward solution

- Disallow $po \cup rf$ 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 pq ∪ 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

- (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]
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

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

\[ \langle x : 0@0 \rangle \]
\[ \langle y : 0@0 \rangle \]
\[ T_1’s \ view \]
\[ \begin{array}{c}
  x \\
  y \\
\end{array} \]
\[ \begin{array}{c}
  0 \\
  0 \\
\end{array} \]
\[ T_2’s \ view \]
\[ \begin{array}{c}
  x \\
  y \\
\end{array} \]
\[ \begin{array}{c}
  0 \\
  0 \\
\end{array} \]

- Global memory is a pool of messages of the form

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

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

**Store-buffering**

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

**Memory**

\[
\begin{array}{c}
\text{T}_1\text{’s view} \\
\langle x : 0@0 \rangle \\
\langle y : 0@0 \rangle \\
\langle x : 1@5 \rangle \\
\end{array}
\]

\[
\begin{array}{c}
\text{T}_2\text{’s view} \\
\langle x : 0@0 \rangle \\
\langle y : 0@0 \rangle \\
\end{array}
\]

- 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

\[ x = y = 0 \]

\[ x = 1; \]

\[ a = y; \text{ // 0} \]

\[ b = x; \text{ // 0} \]

Memory

\[ T_1\text{’s view} \]

\[ x \quad y \]

\[ \langle x : 0@0 \rangle \]

\[ \langle y : 0@0 \rangle \]

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

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

\[ T_2\text{’s view} \]

\[ x \quad y \]

\[ 0 \quad 0 \]

\[ 0 \quad x \]

\[ 5 \quad 5 \]

- 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

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

Memory

\[T_1's \ view\]
\[
\begin{array}{c|c|c}
x & y & \text{timestamp} \\
\hline
0 & 0 & \\
1 & 5 & 5
\end{array}
\]

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

► Global memory is a pool of messages of the form

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

► 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

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

Global memory is a pool of messages of the form

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

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

**Store-buffering**

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

**Coherence Test**

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

**Memory**

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

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

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

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

\[ T_1 \text{’s view} \]
\[ \begin{array}{cc}
  x & y \\
  1 & 0
\end{array} \]
\[ T_2 \text{’s view} \]
\[ \begin{array}{cc}
  x & y \\
  0 & 1
\end{array} \]

Coherence Test

\[ x = 0 \]
\[ x := 1; \quad \]
\[ a = x; \quad // 2 \quad b = x; \quad // 1 \]
\[ \langle x : 0@0 \rangle \]
\[ \langle x : 1@5 \rangle \]

\[ T_1 \text{’s view} \]
\[ \begin{array}{c}
  x \\
  1
\end{array} \]
\[ T_2 \text{’s view} \]
\[ \begin{array}{c}
  x \\
  0
\end{array} \]
Simple operational semantics for C11’s relaxed accesses

Store-buffering

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

Coherence Test

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

Memory

\[
T_1 \text{’s view} \\
\begin{array}{c|c|c}
T_2 \text{’s view} \\
\hline
x & y & 5 \\
0 & 0 & 5 \\
\end{array}
\]

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

\[
\begin{array}{c|c}
T_1 \text{’s view} \\
\hline
x & 5 \\
\end{array}
\]

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

\[
\begin{array}{c|c}
\hline
\langle x : 0@0 \rangle & \langle x : 1@5 \rangle & \langle x : 2@7 \rangle \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c}
\hline
\langle x : 0@0 \rangle & \langle x : 1@5 \rangle & \langle x : 2@7 \rangle \\
\hline
\end{array}
\]

38
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**

\[
x = 0
\]

\[
x := 1;
\]

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

\[
x := 2;
\]

\[
\]

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

\[
T_1’s\ view
\]

\[
\langle x : 0@0 \rangle
\]

\[
\langle y : 0@0 \rangle
\]

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

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

\[
\]

\[
T_2’s\ view
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]
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**

\[ x = 0 \]
\[ x := 1; \]
\[ a = x; \quad //2 \]
\[ x := 2; \]
\[ b = x; \quad //1 \]
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 \textit{“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.
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

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

Load-buffering + dependency

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

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

**Memory**

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

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

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

Must not admit the same execution!
Promises

Load-buffering

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

Load-buffering + dependency

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

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

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

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

\(T_1 \text{ may promise } y := 1, \text{ since it is able to write } y := 1 \text{ by itself.}\)

Load buffering + fake dependency

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

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

\(T_1 \text{ may NOT promise } y := 1, \text{ since it is not able to write } y := 1 \text{ by itself.}\)
Is this behavior possible?

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

Is this behavior possible?

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

No.

Suppose the thread promises \( x := 1 \). Then, once \( a := x \) reads 1, the thread view is increased and so the promise cannot be fulfilled.
Quiz II

Is this behavior possible?

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

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

Yes. And the ARMv7 model allows it!
Quiz II

Is this behavior possible?

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

```
y := x;
```

```
x := y;
```

Yes. And the ARMv7 model allows it!

This behavior can be also explained by sequentialization:

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

```
x := 1;
```

```
\sim
```

```
y := x;
```

```
x := y;
```

```
\sim
```

```
x := 1;
```

```
y := y;
```

```
x := 1;
```

```
y := 1;
```

```
x := y;
```
Is this behavior possible?

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

Yes. And the ARMv7 model allows it!

This behavior can be also explained by sequentialization:

\[
\begin{align*}
    a &:= x; \quad // 1 \\
\sim &
    x &:= 1; \\
    y &:= x; \\
    \quad &
    x &:= y; \\
    \sim &
    y &:= 1; \\
    \quad &
    x &:= y;
\end{align*}
\]
Is this behavior possible?

\[
a := x_{rlx}; \quad \text{\texttt{// 42}}
\]
\[
y := _{rlx} a;
\]
\[
b := y_{rlx};
\]
\[
\text{if } (b = 42)
\]
\[
c := "if";
\]
\[
\text{else}
\]
\[
c := "else";
\]
\[
b := 42;
\]
\[
x := _{rlx} b;
\]
\[
\text{print } (c); \quad \text{\texttt{// prints "if"}}
\]
Is this behavior possible?

Yes. And it can obtained by compiler optimizations!
The full model

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

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)
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)
An intermediate memory model

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

[Podkopaev, L, Vafeiadis POPL’18]
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/