Weak Memory Concurrency in C/C++11

Ori Lahav

November 29, 2017
Example: Dekker’s mutual exclusion

Initially, $x = y = 0$.

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

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 $\sim$ 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; \\
\textbf{if} \ (a = 0) \ \textbf{then} \\
&\text{ /* critical section */}
\end{align*}

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

Is it safe?

Yes, if we assume sequential consistency (SC):

- CPU1 write read
- CPUn ...

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

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

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

Yes, if we assume sequential consistency (SC):

```
CPU 1    ⋯    CPU n
READ     WRITE
Memory
```
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):

CPU 1  ⋯  CPU n
READ  WRITE
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: 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++

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

```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;
}
```
Weak memory models

We look for a *substitute for SC*:

**Unambiguous specification**
- What are the possible outcomes of a multithreaded program?

**Typically called a weak memory model (WMM)**
- Allows more behaviors than SC.

**Amenable to formal reasoning**
- Can prove theorems about the model.
Weak memory models

We look for a substitute for SC:

Unambiguous specification
  ▶ What are the possible outcomes of a multithreaded program?

Typically called a weak memory model (WMM)
  ▶ Allows more behaviors than SC.

Amenable to formal reasoning
  ▶ Can prove theorems about the model.

But it is not easy to get right
  ▶ The Java memory model is flawed.
  ▶ The C/C++11 model is also flawed.
The Problem of Programming Language Concurrency Semantics

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell

University of Cambridge

“Disturbingly, 40+ years after the first relaxed-memory hardware was introduced (the IBM 370/158MP), the field still does not have a credible proposal for the concurrency semantics of any general-purpose high-level language that includes high performance shared-memory concurrency primitives. This is a major open problem for programming language semantics.”

European Symposium on Programming (ESOP) 2015
Plan for rest of the talk

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

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

- **x86-TSO** (2010)
- **POWER** (2011)
- **ARMv8** (2016)
Initially, \( x = y = 0 \).

\[
\begin{align*}
x &:= 1; \\
an &:= y; \quad // 0 \\
y &:= 1; \\
b &:= x; \quad // 0
\end{align*}
\]
Store buffering in x86-TSO

Initially, $x = y = 0$.

$\blacktriangleright \ x := 1;$

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

$\blacktriangleright \ y := 1;$

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

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

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

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

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

- $y := 1$;
- fence;
- $b := x; \quad // 0$
Load buffering in ARM

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

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

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

Initially, $x = y = 0$.

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

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

\[
\begin{align*}
a &:= x; \quad // 1 \\
y &:= 1;
\end{align*}\quad \quad \begin{align*}
b &:= y; \quad // 1 \\
x &:= b;
\end{align*}
\]
Challenge 2: Compilers stir the pot

Initially, $x = y = 0$.

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

$\times$ forbidden under SC

$c := x$; // 0
Initially, $x = y = 0$.

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

\[\times\] forbidden under SC

 compiler optimization

\[\rightarrow\]

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

\[\checkmark\] allowed under SC
Challenge 3: Transformations do not suffice

Program transformations fail short to explain some weak behaviors:

Message passing (MP)

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

Independent reads of independent writes (IRIW)

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

ARM-weak

\[
\begin{align*}
a &:= x; \quad \text{// 1} & y &:= x; \quad \text{// 1} & x &:= y; \quad \text{// 1} \\
x &:= 1;
\end{align*}
\]
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)
The C11 memory model

- Introduced by the ISO C/C++ 2011 standards.
- Defines the semantics of concurrent memory accesses.
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 only
under SC SC behaviors
A spectrum of access modes

- **memory_order_seq_cst**
  - (sc)
  - Full memory fence

- **memory_order_release**
  - Write (rel)
  - No fence (x86); lwsync (PPC)

- **memory_order_acquire**
  - Read (acq)
  - No fence (x86); isync (PPC)

- **memory_order_relaxed**
  - (rlx)
  - No fence

- **Non-atomic (na)**
  - No fence, races are errors

- Explicit primitives for fences
Declarative semantics abstracts away from implementation details.

1. a program \( \sim \) a set of directed graphs (called: execution graphs)
2. The memory model defines what executions are consistent.
3. The semantics of a program is the set of its consistent executions.
4. C/C++11 also has catch-fire semantics (i.e., forbidden data races).
 Execution graphs

Store buffering (SB)

\[
\begin{align*}
x &= y = 0 \\
x &:=_{rlx} 1 & y &:=_{rlx} 1 \\
a &:= y_{rlx} & b &:= x_{rlx}
\end{align*}
\]

Relations

- Program order, \(po\)
- Reads-from, \(rf\)
Figure 2. Semantics of closed program expressions.

\[ \begin{align*}
\#x, \text{hb}(x, x) & \quad \text{(IrreflexiveHB)} \\
\forall x. \text{totalorder}(\{a \in A \mid \text{iswrite}_\ell(a)\}, \text{mo}) \land \text{hb}_\ell \subseteq \text{mo} & \quad \text{(ConsistentMO)} \\
\text{totalorder}((a \in A \mid \text{isSeqCst}(a)), \text{sc}) \land \text{hb}_{\text{SeqCst}} \subseteq \text{sc} \land \text{mo}_{\text{SeqCst}} \subseteq \text{sc} & \quad \text{(ConsistentSC)} \\
\forall b. \text{rf}(b) \neq \perp & \quad \text{rf}(b) \neq \perp \iff \exists \ell. \text{iswrite}_\ell(a) \land \text{isread}_\ell(b) \land \text{hb}(a, b) & \quad \text{(ConsistentRFdom)} \\
\forall a, b. \text{rf}(b) = a & \quad \forall a, b. \text{rf}(b) = a \iff \exists \ell. \text{iswrite}_\ell(a) \land \text{isread}_\ell(b) \land \neg \text{hb}(a, b) \\
\forall a, b. \text{rf}(b) = a \land (\text{mode}(a) = \text{na} \lor \text{mode}(b) = \text{na}) & \quad \Rightarrow \text{hb}(a, b) & \quad \text{(ConsistentRFna)} \\
\forall a, b. \text{rf}(b) = a \land \text{isSeqCst}(b) & \quad \Rightarrow \text{isc}(a, b) \lor \neg \text{isSeqCst}(a) \land (\forall x. \text{isc}(x, b) \Rightarrow \neg \text{hb}(a, x)) & \quad \text{(RestrSCRReads)} \\
\forall a, b. \text{hb}(a, b) \land \text{mo}(\text{rf}(b), \text{rf}(a)) \land \text{locs}(a) = \text{locs}(b) & \quad \text{(CoherentRR)} \\
\forall a, b. \text{hb}(a, b) \land \text{mo}(\text{rf}(b), \text{rf}(a)) \land \text{iswrite}(a) \land \text{locs}(a) = \text{locs}(b) & \quad \text{(CoherentWR)} \\
\forall a, b. \text{hb}(a, b) \land \text{mo}(\text{rf}(b), \text{rf}(a)) \land \text{iswrite}(b) \land \text{locs}(a) = \text{locs}(b) & \quad \text{(CoherentRW)} \\
\forall a. \text{isrmw}(a) \land \text{rf}(a) \neq \bot & \quad \Rightarrow \text{mo}(\text{rf}(a), a) \land \#c. \text{mo}(\text{rf}(a), c) \land \text{mo}(a, c) & \quad \text{(AtomicRMW)} \\
\forall a, b, \ell. \text{lab}(a) = \text{lab}(b) = A(\ell) & \quad \Rightarrow a = b & \quad \text{(ConsistentAlloc)}
\end{align*} \]

where \(\text{iswrite}_\ell(a) \equiv \exists X, \text{vold}. \text{lab}(a) \in \{\text{WX}(\ell, v), \text{RMWX}(\ell, \text{vold}, v)\} \quad \text{iswrite}(a) \equiv \exists v. \text{iswrite}_\ell(a)(v) \quad \text{isc}(a, b) \equiv \exists \ell. \text{isc}(a, b)(\ell) \quad \text{etc.} \quad \text{isc}(a, b) \equiv \exists \ell. \text{isc}(a, b)(\ell) \quad \text{iswrite}(a) \equiv \exists v. \text{iswrite}_\ell(a)(v) \quad \text{isc}(a, b) \equiv \exists \ell. \text{isc}(a, b)(\ell) \equiv \exists v. \text{isc}(a, b)(v) \quad \text{etc.} \quad \text{isc}(a, b) \equiv \exists \ell. \text{isc}(a, b)(\ell) \equiv \exists v. \text{isc}(a, b)(v) \quad \text{etc.} \quad \text{isc}(a, b) \equiv \exists \ell. \text{isc}(a, b)(\ell) \equiv \exists v. \text{isc}(a, b)(v) \quad \text{etc.} \quad \text{isc}(a, b) \equiv \exists \ell. \text{isc}(a, b)(\ell) \equiv \exists v. \text{isc}(a, b)(v) \quad \text{etc.} \end{align*} \]

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

\begin{align*}
c : \text{W}(\ell, 1) \xrightarrow{\text{rf}} a & : \text{R}(\ell, 1) & \text{c : W}(\ell, 2) \xrightarrow{\text{mo}} a & : \text{W}(\ell, 1) & \text{c : W}(\ell, 1) \xrightarrow{\text{rf}} a & : \text{R}(\ell, 1) \\
\mid \text{rf} \downarrow \text{hb} \quad \uparrow \text{mo} & \mid \text{rf} \downarrow \text{hb} \quad \downarrow \text{mo} & \mid \text{rf} \downarrow \text{hb} \\
d : \text{W}(\ell, 2) \xrightarrow{\text{rf}} b & : \text{R}(\ell, 2) & \text{d : W}(\ell, 2) \xrightarrow{\text{rf}} b & : \text{R}(\ell, 2) & \text{d : W}(\ell, 2) \xrightarrow{\text{rf}} b & : \text{W}(\ell, 2) \\
\text{violates CoherentRR} & \text{violates CoherentWR} & \text{violates CoherentWR} & \text{violates CoherentWR} & \text{violates CoherentWR} & \text{violates CoherentWR}
\end{align*} 

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
Definition (Declarative definition of SC)

$G$ is **SC-consistent** if there exists a relation $sc$ s.t. the following hold:

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

Definition (SC-per-location)

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

- $sc_x$ is a total order on the events of $G$ that access $x$.
- If $po \cup rf \subseteq sc_x$.
- If $\langle a, b \rangle \in rf$ then there does not exist $c \in W_x$ such that $\langle a, c \rangle \in sc_x$ and $\langle c, b \rangle \in sc_x$. 
\[
x = 0 \\
x :=_{rlx} 1 \quad \| \quad x :=_{rlx} 2 \\
a := x_{rlx} \quad \| \quad b := x_{rlx}
\]

Inconsistent!
Release/acquire synchronization

SC-per-location is often too weak:

▶ It does not support the message passing idiom:

Message passing (MP)

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

▶ We cannot even implement locks:

Simple lock

\[
\begin{align*}
\text{lock}(); & \quad \text{lock}(); \\
\text{unlock}(); & \quad \text{unlock}(); \\
x &:= 1; & a &:= x; \quad \text{// 1} \\
x &:= 2;
\end{align*}
\]
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

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

1 race 2
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 == 1){
      x = 1;
      print(y);
   }

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

   fence rel;
   fence acq;
Synchronization in C/C++11 through examples

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

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

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

1. int y = 0;
   int x = 0;
y = 42;  // if(x == 1) {
  x = 1,  \textcolor{red}{\textit{race}}  \quad \text{print}(y);
} 

2. int y = 0;
   \text{atomic<int>} x = 0;
y = 42;  // if(x_{rlx} == 1) {
  x =_{rlx} 1;  \textcolor{red}{\textit{race}}  \quad \text{print}(y);
} 

3. int y = 0;
   \text{atomic<int>} x = 0;
y = 42;  // if(x_{acq} == 1) {
  x =_{rel} 1;  \textcolor{red}{\textit{acq}}  \quad \textcolor{green}{\textit{rf}}  \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);
   } // 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; // rf
     print(y);
   } // sw
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);
}
```

3

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

4

```c
int y = 0;
atomic<int> x = 0;
y = 42; || if(x_rlx == 1) {
  fence_rlx; fence_acq;
  x = rlx 1; print(y);
}
```
Synchronization in C/C++11 through examples

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

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

3. int y = 0;
   atomic<int> x = 0;
   y = 42; // if(xacq == 1) {
     x = _rel 1; // rel, sw
     print(y);
   }

4. int y = 0;
   atomic<int> x = 0;
   y = 42; // if(xrlx == 1) {
     fence_rel;
     fence_acq;
     x = rlx 1;    // race
     print(y);
   }
Synchronization in C/C++11 through examples

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

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

3. int y = 0;
   atomic<int> x = 0;
   y = 42;  // if(x_acq == 1) {
   x = x_acq 1;  // rf
   print(y);
   }

4. int y = 0;
   atomic<int> x = 0;
   y = 42;  // if(x_rlx == 1) {
   fence_x_rlx;  // rf
   fence_x_acq;
   x = x_rlx 1;  // race
   print(y);
   }
The “happens-before” relation

Definition (happens-before)

- \( a \xrightarrow{\text{po}} b \) should be acyclic.
- The SC-per-location orders should contain \( \text{hb} \).
- Using acquire CAS’s and release writes, we can implement locks.
How to guarantee only SC behaviors (i.e., \( a = 1 \lor b = 1 \))?
SC semantics

- Perhaps surprisingly, the semantics of SC atomics is the most complicated part of the model.

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

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

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

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

- Recently, the standard committee fixed the specification following:
  [Repairing Sequential Consistency in C/C++11 PLDI’17]
The “out-of-thin-air” problem
C/C++11 is too weak

<table>
<thead>
<tr>
<th>non-atomic</th>
<th>relaxed</th>
<th>release/acquire</th>
<th>sc</th>
</tr>
</thead>
</table>
C/C++11 is too weak

non-atomic □ relaxed □ release/acquire □ sc

Load-buffering

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

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

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

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

\[
\begin{align*}
[x = y = 0] \\
R \times 1 & \quad R \times 1 \quad \text{program order} \\
W \times 1 & \quad W \times 1
\end{align*}
\]
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!**

\[
[x = y = 0]
\]

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

program order
C/C++11 is too weak

non-atomic  □  relaxed  □  release/acquire  □  sc

Load-buffering

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

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

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

\[
[x = y = 0]
\]

program order
reads from

\[
\begin{align*}
    R \times 1 &\quad R \times 1 \\
    W \times 1 &\quad W \times 1 \\
    R \times 1 &\quad R \times 1
\end{align*}
\]
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!**

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

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

program order

reads from
C/C++11 is too weak

non-atomic ☐ relaxed ☐ release/acquire ☐ sc

Load-buffering

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

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

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

Load-buffering + data dependency

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

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

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

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

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

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

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

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

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

program order

reads from

syntactic dependency

\[ x = y = 0 \]
The hardware solution

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

### Load-buffering

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

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

### Load-buffering + data dependency

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

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

### Load-buffering + fake dependency

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

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

This approach is not suitable for a programming language: **Compilers do not preserve syntactic dependencies.**
The “out-of-thin-air” problem

- The C/C++11 model is too weak:
  - Values might appear *out-of-thin-air*.
  - The *DRF guarantee* is broken.

- A straightforward solution:
  - Disallow \( po \cup rf \) cycles
  - But, on weak hardware it carries a certain *implementation cost*.

- Solving the problem without changing the compilation schemes will require a *major revision* of the standard.
A ‘promising’ solution to OOTA

[Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, Derek Dreyer  POPL’17]

We propose a model that satisfies all WMM desiderata, and covers nearly all features of C11.

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

**Key idea:** Start with an operational interleaving semantics, but allow threads to promise to write in the future.
Simple operational semantics for C11’s relaxed accesses

Store-buffering

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

Store-buffering

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

Memory

\[
T_1\'s \ view \quad T_2\'s \ view
\]

\[
\begin{array}{c|c}
x & y \\ \hline
0 & 0 \\
0 & 0
\end{array}
\]

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

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

Memory

\begin{align*}
T_1’s \text{ view} & :\begin{array}{c}
\langle x : 0@0 \rangle \\
\langle y : 0@0 \rangle \\
\langle x : 1@1 \rangle \\
\rangle \\
\end{array} \\
\begin{array}{c|c}
x & y \\
\hline
x & 0 \times \\
\hline
1 & 0 \\
\end{array} \\
T_2’s \text{ view} & :\begin{array}{c}
\langle x : 0@0 \rangle \\
\langle y : 0@0 \rangle \\
\langle x : 1@1 \rangle \\
\rangle \\
\begin{array}{c|c}
x & y \\
\hline
0 & 0 \\
\end{array} \\
\end{array}
\end{align*}

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

- Each thread maintains a \textit{thread-local view} recording the last observed timestamp for every location
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.
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.
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

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

Memory

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

T_2's view

\[
\begin{array}{c|c|c}
T_2's \ view & x & y \\
\hline
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{array}
\]

Coherence Test

\[
\begin{align*}
x &= 0 \\
x &= 1; & x &= 2; \\
a &= x; & /\ 2 & b &= x; & /\ 1
\end{align*}
\]
### Simple operational semantics for C11’s relaxed accesses

#### Store-buffering

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

#### Coherence Test

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

#### Memory

<table>
<thead>
<tr>
<th>Store-buffering</th>
<th>T₁’s view</th>
<th>T₂’s view</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle x : 0@0 \rangle)</td>
<td>(x) (y)</td>
<td>(0) (0)</td>
</tr>
<tr>
<td>(\langle y : 0@0 \rangle)</td>
<td>(x) (y)</td>
<td>(0) (0)</td>
</tr>
<tr>
<td>(\langle x : 1@1 \rangle)</td>
<td>(x) (y)</td>
<td>(1) (1)</td>
</tr>
<tr>
<td>(\langle y : 1@1 \rangle)</td>
<td>(x) (y)</td>
<td>(1) (1)</td>
</tr>
</tbody>
</table>

#### T₁’s view

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>(0)</td>
</tr>
</tbody>
</table>

#### T₂’s view

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>(0)</td>
</tr>
</tbody>
</table>

#### Coherence Test

<table>
<thead>
<tr>
<th>Coherence Test</th>
<th>T₁’s view</th>
<th>T₂’s view</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle x : 0@0 \rangle)</td>
<td>(x)</td>
<td>(0)</td>
</tr>
</tbody>
</table>

#### T₁’s view

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>(0)</td>
</tr>
</tbody>
</table>

#### T₂’s view

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>(0)</td>
</tr>
</tbody>
</table>
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

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

**Coherence Test**

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

\[ x = 0 \]
\[ x := 1; \quad \text{∥ 2} \]
\[ a = x; \quad \text{∥ 2} \]
\[ x := 2; \quad \text{∥ 2} \]
\[ b = x; \quad \text{∥ 1} \]

Memory

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

\[ T_1's \ view \]
\[ x \quad y \]
\[ \text{∥ 0} \quad \text{∥ 0} \]
\[ \text{∥ 1} \quad \text{∥ 0} \]

\[ T_2's \ view \]
\[ x \quad y \]
\[ \text{∥ 0} \quad \text{∥ 0} \]
\[ \text{∥ 1} \quad \text{∥ 1} \]
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

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

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

**Coherence Test**

\[ x = 0 \]
\[ x := 1; \]
\[ a = x; \ // 2 \]

\[ x := 2; \]
\[ b = x; \ // 1 \]
### Simple operational semantics for C11’s relaxed accesses

#### Store-buffering

<table>
<thead>
<tr>
<th>x = y = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
</tr>
<tr>
<td>a = y; // 0</td>
</tr>
<tr>
<td>y = 1;</td>
</tr>
<tr>
<td>b = x; // 0</td>
</tr>
</tbody>
</table>

#### Coherence Test

<table>
<thead>
<tr>
<th>x = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1;</td>
</tr>
<tr>
<td>a = x; // 2</td>
</tr>
<tr>
<td>x := 2;</td>
</tr>
<tr>
<td>b = x; // 1</td>
</tr>
</tbody>
</table>

---

#### Memory

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁’s view</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T₂’s view</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁’s view</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T₂’s view</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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.
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

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

Load-buffering + dependency

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

Memory

\[
\begin{align*}
\langle x : 0@0 \rangle \\
\langle y : 0@0 \rangle \\
\langle y : 1@1 \rangle \\
\langle x : 1@1 \rangle \\
\end{align*}
\]

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

\[
\begin{array}{c|c}
T_2's \ view \\
\hline
x & y \\
\hline
1 & 1 \\
\hline
\end{array}
\]

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.

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

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

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

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

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

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

No.

Suppose the thread promises \( x := 1 \). Then, once \( a := x \) reads 1, the thread view is increased and so the promise cannot be fulfilled.
Is this behavior possible?

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

Is this behavior possible?

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

```
y := x;
```

```
x := y;
```

Yes. And the ARM model allows it!
Quiz

Is this behavior possible?

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

Yes. And the ARM model allows it!

This behavior can be also explained by sequentialization:

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

\[
\begin{align*}
a &:= x; \quad // 1 \\
x &:= 1; \\
y &:= x; \\
x &:= y;
\end{align*}
\]
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
- Efficient h/w mappings (x86-TSO, Power, ARM)
- Compiler optimizations (incl. reorderings, eliminations)
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
▶ Efficient h/w mappings (x86-TSO, Power, ARM)
▶ Compiler optimizations (incl. reorderings, eliminations)
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 \textit{locally fixable}.
- But ruling out OOTA requires an entirely different approach.

The \textit{promising model} may be the solution.

\textit{Thank you!}

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