Weak Memory Concurrency in C/C++11

Ori Lahav

Haifa::C++ meeting
November 21, 2017
Example: Dekker’s mutual exclusion

Initially, $x = y = 0$.

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

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

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

- cpu 1 write
- 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?
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):

Yes, if we assume sequential consistency (SC):

CPU 1  ...  CPU n

READ  WRITE

Memory

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, \( x = y = 0 \).

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

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

Is it safe?

Yes, if we assume sequential consistency (SC):

No existing hardware implements SC!

- SC is very expensive (memory \(~100\) 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;
}
```
int X, Y, a, b;

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

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

int main () {
    int cnt = 0;
    do {
        X = 0; Y = 0;
        thread first(thread1);
        thread second(thread2);
        first.join();
        second.join();
        cnt++;
    } while (a != 0 || b != 0);

    printf("%d\n",cnt);
    return 0;
}

If Dekker’s mutual exclusion is safe, this program will not terminate
Weak memory models

We look for a substitute for SC:

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

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

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

POWER (2011)

ARMv8 (2016)
Store buffering in x86-TSO

Initially, $x = y = 0$.

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

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

\[
\begin{align*}
\text{CPU 1:} & \quad x := 1; \\
\text{CPU 2:} & \quad y := 1; \\
\text{COPY:} & \quad a := y; // 0 \\
\text{COPY:} & \quad b := x; // 0 \\
\end{align*}
\]
Initially, \( x = y = 0 \).

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

$x := 1$;  
\textbf{fence};  
a := y;  // 0  

$y := 1$;  
\textbf{fence};  
b := x;  // 0  

\[
\begin{array}{c}
\text{CPU 1} \\
\text{READ} \\
\downarrow \\
x := 1 \\
\downarrow \\
\text{WRITE-BACK} \\
\downarrow \\
x \leftrightarrow 0 \\
\end{array}
\quad
\begin{array}{c}
\text{CPU 2} \\
\text{WRITE} \\
\downarrow \\
y := 1 \\
\downarrow \\
y \leftrightarrow 0 \\
\end{array}
\]
Load buffering in ARM

Initially, $x = y = 0$.

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

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

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

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

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

Initially, $x = y = 0$.

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

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

\[\times\] forbidden under SC
Initially, $x = y = 0$. 

- Forbidden under SC: 
  
  ```
  x := 1;
  y := 1;
  a := x;
  b := y;  // 1
  c := x;  // 0
  ```

- Allowed under SC: 
  
  ```
  x := 1;
  y := 1;
  a := x;
  b := y;  // 1
  c := a;  // 0
  ```
**Challenge 3: Transformations do not suffice**

Program transformations fail short to explain some weak behaviors:

<table>
<thead>
<tr>
<th>Message passing (MP)</th>
<th>Independent reads of independent writes (IRIW)</th>
<th>ARM-weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := 1; \parallel a := y; \ // 1$</td>
<td>$\parallel c := y; \ // 1$</td>
<td>$a := x; \ // 1$</td>
</tr>
<tr>
<td>$y := 1; \parallel b := x; \ // 0$</td>
<td>$\parallel d := x; \ // 0$</td>
<td>$x := 1;$</td>
</tr>
<tr>
<td>$\parallel y := 1;$</td>
<td>$\parallel x := y; \ // 1$</td>
<td>$y := x; \ // 1$</td>
</tr>
</tbody>
</table>
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.
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 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)

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

Relations

- Program order, \( po \)
- Reads-from, \( rf \)
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
SC-per-location

**Definition (Declarative definition of SC)**

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

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

**Definition (SC-per-location)**

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

- $\text{sc}_x$ is a total order on the events of $G$ that access $x$.
- If $\text{po} \cup \text{rf} \subseteq \text{sc}_x$.
- If $\langle a, b \rangle \in \text{rf}$ then there does not exist $c \in W_x$ such that $\langle a, c \rangle \in \text{sc}_x$ and $\langle c, b \rangle \in \text{sc}_x$. 
SC-per-location: Example

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

program order reads from inconsistent!
Release/acquire synchronization

SC-per-location is often too weak:

- It does not support the message passing idiom:

```
Message passing (MP)
y := 42;  a := x;  // 1
x := 1;   b := y;  // 0
```

- We cannot even implement locks:

```
Simple lock
lock();
x := 1;
x := 2;
unlock();  lock();
a := x;    // 1
unlock();
```
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); }  // fence
```

1

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

2

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

3
int y = 0;
int x = 0;
y = 42;  // if(x == 1){
x = 1,  // race
}
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_acq == 1){
     x = rel;
     fence;
     fence;
     x = rlx 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, // print(y);
} // race
```

2

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

2

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

3

```c
int y = 0;
atomic<int> x = 0;
y = 42;  // if(x_{acq} == 1){
x =_{rel} 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
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 == 1) {
x = 1;    // rf
} 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 == 1){
     x = 1;  // race
   }  // print(y);
3. int y = 0;
   atomic<int> x = 0;
   y = 42;  // if(x == 1){
     x = 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, // 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
   print(y);
   }```

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

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

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

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

4. `int y = 0;`  
   `atomic<int> x = 0;`  
   `y = 42;`  
   `if(x_rlx == 1){`  
   `fence_rel;`  
   `fence_acq;`  
   `x = 1;`  
   `print(y);`  
   `}`
The “happens-before” relation

Definition (happens-before)

- \( a \xrightarrow{\text{po}} b \) \( a \xrightarrow{\text{rf}} b \) \( a \xrightarrow{\text{hb}} b \) \( a \xrightarrow{\text{hb}} c \) \( b \xrightarrow{\text{hb}} c \)

- \( \text{hb} \) should be acyclic.
- The SC-per-location orders should contain \( \text{hb} \).
- Using acquire CAS’s and release writes, we can implement locks.
SC accesses and fences

**Store buffer**

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

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

\[
\begin{align*}
x & :=_{sc} 1; \\
a & := y_{sc}; \\
y & :=_{sc} 1; \\
b & := x_{sc}; \\
\Rightarrow \quad x & :=_{rlx} 1; \\
\text{fence}_{sc}; \\
y & :=_{rlx} 1; \\
a & := y_{rlx}; \\
b & := x_{rlx};
\end{align*}
\]
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!)

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

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

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

- 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

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

non-atomic □ relaxed □ release/acquire □ sc

Load-buffering

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

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

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

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

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

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

\[
[x = y = 0]
\]

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

program order
C/C++11 is too weak

non-atomic □ relaxed □ release/acquire □ sc

Load-buffering

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

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

\[
[x = y = 0]
\]

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

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

Load-buffering

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

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

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

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

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

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

C/C++11 allows this behavior.

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

Load-buffering + control dependency

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

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.

---

**Load-buffering**

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

**Load-buffering + data dependency**

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

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

### Load-buffering

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a := x; // 1</code></td>
<td></td>
</tr>
<tr>
<td><code>y := 1;</code></td>
<td></td>
</tr>
<tr>
<td><code>b := y; // 1</code></td>
<td></td>
</tr>
<tr>
<td><code>x := b;</code></td>
<td></td>
</tr>
</tbody>
</table>

### Load-buffering + data dependency

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a := x; // 1</code></td>
<td></td>
</tr>
<tr>
<td><code>y := a;</code></td>
<td></td>
</tr>
<tr>
<td><code>b := y; // 1</code></td>
<td></td>
</tr>
<tr>
<td><code>x := b;</code></td>
<td></td>
</tr>
</tbody>
</table>

### Load-buffering + fake dependency

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a := x; // 1</code></td>
<td></td>
</tr>
<tr>
<td><code>y := a + 1 - a;</code></td>
<td></td>
</tr>
<tr>
<td><code>b := y; // 1</code></td>
<td></td>
</tr>
<tr>
<td><code>x := b;</code></td>
<td></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

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

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

[Jeheeon 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.
## Store-buffering

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

Store-buffering

\[ x = y = 0 \]

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

Memory

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

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

- 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

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

Memory

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

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

\[T_2\text{'s view} \]
\[
x \quad y
\]
\[
\text{0}
\]
\[
\text{0}
\]

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

\[T_2\text{'s view} \]
\[
x \quad y
\]
\[
\text{0}
\]
\[
\text{0}
\]

- 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
Simple operational semantics for C11’s relaxed accesses

Store-buffering

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

Memory

\[
\begin{array}{ll}
T_1’s \ view \\
\langle x : 0@0 \rangle \\
\langle y : 0@0 \rangle \\
\langle x : 1@1 \rangle \\
\langle y : 1@1 \rangle \\
\end{array}
\]

\[
\begin{array}{ll}
T_2’s \ view \\
\langle x : 0@0 \rangle \\
\langle y : 0@0 \rangle \\
\langle x : 1@1 \rangle \\
\langle y : 1@1 \rangle \\
\end{array}
\]

- Global memory is a pool of messages of the form

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

- Each thread maintains a \textit{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

\( \langle x : 0@0 \rangle \)
\( \langle y : 0@0 \rangle \)
\( \langle x : 1@1 \rangle \)
\( \langle y : 1@1 \rangle \)

\( T_1 \)'s view

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

\( T_2 \)'s view

\[ \begin{array}{cc}
  x & y \\
  0 & 1 \\
\end{array} \]

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

- 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

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

Memory

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

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

- Global memory is a pool of messages of the form

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

- 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

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

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

Coherence Test

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

**Store-buffering**

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

**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**

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

**Coherence Test**

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

**Memory**

- \( T_1 \)’s view:
  - \( x: 0@0 \) → 0
  - \( y: 0@0 \) → 0

- \( T_2 \)’s view:
  - \( x: 1@1 \) → 1
  - \( y: 1@1 \) → 1
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

\[
\begin{align*}
    x &= y = 0 \\
    x &= 1; & y &= 1; \\
    a &= y; & b &= x; \\
    \\
    T_1's \ view &\quad T_2's \ view \\
    \langle x : 0@0 \rangle &\quad \langle x : 1@1 \rangle \quad \langle x : 2@2 \rangle \\
    \langle y : 0@0 \rangle &\quad \langle y : 1@1 \rangle \\
    \langle x : 1@1 \rangle &\quad \langle y : 1@1 \rangle \\
\end{align*}
\]

**Coherence Test**

\[
\begin{align*}
    x &= 0 \\
    x &:= 1; & x &:= 2; \\
    a &= x; & b &= x; \\
    \\
    T_1's \ view &\quad T_2's \ view \\
    \langle x : 0@0 \rangle &\quad \langle x : 1@1 \rangle \quad \langle x : 2@2 \rangle \\
    \langle x : 1@1 \rangle &\quad \langle x : 1@1 \rangle \\
\end{align*}
\]
Simple operational semantics for C11’s relaxed accesses

**Store-buffering**

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

**Coherence Test**

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

**Store-buffering**

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

\[ T_1's \ view \]
\[ \langle x : 0@0 \rangle \]
\[ \langle y : 0@0 \rangle \]
\[ \langle x : 1@1 \rangle \]
\[ \langle y : 1@1 \rangle \]

\[ T_2's \ view \]
\[ \langle x : 0@0 \rangle \]
\[ \langle y : 0@0 \rangle \]

---

**Coherence Test**

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

\[ T_1's \ view \]
\[ \langle x : 0@0 \rangle \]
\[ \langle x : 1@1 \rangle \]
\[ \langle x : 2@2 \rangle \]

\[ T_2's \ view \]
\[ \text{No coherence test shown} \]

---
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.
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{\(\mathbf{1}\)} \\
  y & := 1; \quad \text{\(\mathbf{1}\)} \\
  x & := y;
\end{align*}
\]

Memory

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

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

\[
\begin{array}{cccc}
T_2 \text{’s view} & x & y \\
\hline
x & 1 \\
y & 1 \\
x & 1 \\
\end{array}
\]

- 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

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

Load-buffering + dependency

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

Memory

<table>
<thead>
<tr>
<th>Memory</th>
<th>(T_1)'s view</th>
<th>(T_2)'s view</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle x : 0@0 \rangle)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>(\langle y : 0@0 \rangle)</td>
<td>(y)</td>
<td>(y)</td>
</tr>
<tr>
<td>(\langle y : 1@1 \rangle)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(\langle x : 1@1 \rangle)</td>
<td>(1)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

Must not admit the same execution!
Promises

Load-buffering

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

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

Key Idea

A thread can only promise if it can perform the write anyway (even without having made the promise)

Load-buffering + dependency

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

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

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

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

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

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

\[
a := x; \quad // 1
\]
\[
x := 1;
\]
Is this behavior possible?

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

**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?

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

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

Yes. And the ARM model allows it!
Quiz

Is this behavior possible?

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

Yes. And the ARM model allows it!

This behavior can be also explained by sequentialization:

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