Weak Memory Concurrency in Programming Languages

Ori Lahav

February, 2019
Example: Dekker’s mutual exclusion

Initially, $x = y = 0$.

```
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

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, \( x = y = 0 \).

\[
x := 1;
\]

\[
a := y;
\]

\[
\text{if } (a = 0) \text{ then}
\]

\[
/* \text{critical section} */
\]

\[
y := 1;
\]

\[
b := x;
\]

\[
\text{if } (b = 0) \text{ then}
\]

\[
/* \text{critical section} */
\]

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, $x = y = 0$.

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

if ($a = 0$) then

/* critical section */

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

if ($b = 0$) then

/* critical section */

Is it safe?

Yes, if we assume sequential consistency (SC):

Yes, if we assume sequential consistency (SC):
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):

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++

```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.
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
Challenge 1: Various hardware models

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

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

The diagram illustrates the memory buffering in x86-TSO, where CPU 1 reads from memory, write-backs to memory, and CPU 2 reads from memory.
Initially, $x = y = 0$.

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

$$
x := 1; \quad \text{⇒} \quad a := y; \quad \text{// 0}
$$

$$
y := 1; \quad \text{⇒} \quad b := x; \quad \text{// 0}
$$
Store buffering in x86-TSO

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

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

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

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

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

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

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

✗ forbidden under SC
Challenge 2: Compilers stir the pot

Initially, $x = y = 0$.

Initially,

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

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

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

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

\[\text{\texttimes \ forbidden under SC} \quad \checkmark \quad \text{allowed under SC}\]
Challenge 3: Transformations do not suffice

Program transformations fail short to explain some weak behaviors:

**Message passing (MP)**

```plaintext
x := 1;  // 1
y := 1;  // 0
a := y;  // 1
b := x;  // 0
```

**Independent reads of independent writes (IRIW)**

```plaintext
a := x;  // 1
b := y;  // 0
x := 1;  // 1
y := 1;  // 0
c := y;  // 1
d := x;  // 0
```

**ARM-weak**

```plaintext
a := x;  // 1
x := 1;  // 1
y := x;  // 1
x := y;  // 1
```
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 $\leadsto$ 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, \( rf \)
\[ \\{ \llbracket v,v \rrbracket \in \mathbb{Z} \} \times \{ \llbracket v,v \rrbracket \in \mathbb{Z} \} \times \{ \llbracket v,v \rrbracket \in \mathbb{Z} \} \rightarrow \{ a \in \mathbb{N} \} \]
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 \( sc \) 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 \( \langle c, b \rangle \in sc_x \).
SC-per-location: Example

\[ x = 0 \]
\[ x :=_{rlx} 1 \parallel x :=_{rlx} 2 \]
\[ a := x_{rlx} \parallel b := x_{rlx} \]

Inconsistent!
Release/acquire synchronization

SC-per-location is often too weak:

- It does not support the message passing idiom:

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

- We cannot even implement locks:

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

1 race 22
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;  // 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 == 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);
}
```

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;    // 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(xrlx == 1) {
     x =rlx 1;  // race
                
   }      // print(y);

3. int y = 0;
   atomic<int> x = 0;
   y = 42;  // if(xacq == 1) {
     x =rel 1;  // rf
                
   }      // print(y);
Synchronization in 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 == 1){
x = 1; print(y);
}
```

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

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

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

3. int y = 0;
   atomic<int> x = 0;
   y = 42;  // if(x_{acq} == 1){
      \quad \text{fence}_{rel}; \text{fence}_{acq};
      \quad \text{fence}_{rel};  \text{fence}_{acq};
      \quad x =_{rlx} 1;  \text{race}
       \quad \text{print}(y);
     }$

4. int y = 0;
   atomic<int> x = 0;
   y = 42;  // if(x_{rlx} == 1){
      fence_{rel};  \text{fence}_{acq};
      x =_{rlx} 1;  \text{race}
       \quad \text{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;
    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;  // rf
   print(y);  // sw
   }
```  

4. ```
   int y = 0;
   atomic<int> x = 0;
   y = 42;  // if(xrlx == 1){
   fence_rel;  // rf
   fence_acq;  // sw
   x =rlx 1;  // race
   print(y);  //
   ```
The “happens-before” relation

**Definition (happens-before)**

\[
\begin{align*}
\text{a: } & W \text{ Rel } \\
\text{b: } & R \text{ Acq }
\end{align*}
\]

\[
\begin{array}{c}
\text{a} \xrightarrow{\text{po}} \text{b} \\
\text{a} \xrightarrow{\text{rf}} \text{b} \\
\text{a} \xrightarrow{\text{hb}} \text{b} \\
\text{a} \xrightarrow{\text{hb}} \text{b} \\
\text{a} \xrightarrow{\text{hb}} \text{c} \\
\text{a} \xrightarrow{\text{hb}} \text{c}
\end{array}
\]

- **hb** should be acyclic.
- The SC-per-location orders should contain **hb**.
- Using acquire CAS’s and release writes, we can implement locks.
SC accesses and fences

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

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

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

Store buffer
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!)

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

- 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

<table>
<thead>
<tr>
<th>non-atomic</th>
<th>relaxed</th>
<th>release/acquire</th>
<th>sc</th>
</tr>
</thead>
</table>

**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!**
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>

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!**

\[
[x = y = 0]
\]

\[
\begin{aligned}
R \times 1 & \quad R \times y 1 \\
W y 1 & \quad W \times x 1
\end{aligned}
\]

program order
C/C++11 is too weak

non-atomic □ relaxed □ release/acquire □ sc

Load-buffering

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

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

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

\[
[x = y = 0]
\]

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

program order
C/C++11 is too weak

non-atomic □ relaxed □ release/acquire □ sc

Load-buffering

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

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

\[
[x = y = 0]
\]

\[
\begin{array}{c}
\text{R}_x 1 \\
\text{W}_y 1 \\
\text{W}_x 1 \\
\text{R}_y 1
\end{array}
\]

program order

reads from
C/C++11 is too weak

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

Load-buffering

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

C/C++11 allows this behavior because \textbf{POWER \& ARM allow it!}

[\[ x = y = 0 \]]

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

program order
C/C++11 is too weak

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

Load-buffering

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

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

Load-buffering + data dependency

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

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

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

C/C++11 allows this behavior.

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

non-atomic  □  relaxed  □  release/acquire  □  sc

Load-buffering + control dependency

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

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.

### Load-buffering

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

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

### Load-buffering + data dependency

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

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

\([x = y = 0]\)

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

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

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

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

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

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

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

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

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

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

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

Memory

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

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

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

- 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

\( x = y = 0 \)

\[ x = 1; \quad y = 1; \]

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

Global memory is a pool of messages of the form

\( \langle \text{location} : \text{value}@\text{timestamp} \rangle \)

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

\[ T_1's \ view \]

\( x \quad y \)

\( \begin{array}{cc}
0 & 0 \\
\hline
1 & 1
\end{array} \)

\[ T_2's \ view \]

\( x \quad y \)

\( \begin{array}{cc}
0 & 0 \\
\hline
1 & 1
\end{array} \)
Simple operational semantics for C11’s relaxed accesses

### Store-buffering

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

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

### Coherence Test

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

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

**Store-buffering**

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

**Coherence Test**

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

**Store-buffering**

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

**Coherence Test**

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

**Memory**

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

**T_2’s view**

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

**Store-buffering**

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

**Coherence Test**

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

**Store-buffering**

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

**Coherence Test**

\[
\begin{align*}
  x &= 0 \\
  x &= 1; \\ 
  a &= x; \quad // 2 \\
  x &= 2; \\ 
  b &= x; \quad // 1
\end{align*}
\]
To model load-store reordering, we allow “promises”.

At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.
To model load-store reordering, we allow “promises”.

At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.
To model load-store reordering, we allow “promises”.

At any point, a thread may promise to write a message in the future, allowing other threads to read from the promised message.
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

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

### Load-buffering + dependency

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

---

**Memory**

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

**\(T_1\)’s view**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**\(T_2\)’s view**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

---

**Must not admit the same execution!**
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
\[
\begin{align*}
a &:= x; \quad // 1 \\
y &:= 1;
\end{align*}
\]
\[
\begin{align*}
x &:= y;
\end{align*}
\]

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

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

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

\(T_1\) **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?

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

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

Yes. And the ARM model allows it!
Is this behavior possible?

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

Yes. And the ARM model allows it!

This behavior can be also explained by sequentialization:

\[
\begin{align*}
a &:= x; \quad // 1 \\
x &:= 1; \quad \quad y := x; \quad x := y; \quad \sim \quad x := 1; \quad a := x; \quad // 1 \\
y &:= x; \quad y := x; \quad x := y;
\end{align*}
\]
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) [POPL’19]
- 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) [POPL’19]
▶ Compiler optimizations (incl. reorderings, eliminations)
Challenge: Verification

How to verify the correctness of programs running under weak memory semantics?

- How to explore runs of the “promising semantics”?
- What memory semantics allow *decidable* verification?
- How to verify the *robustness* of programs against a weak memory model?
- Can we utilize weak memory semantics for *easier* model checking?
Summary

Thank you!

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

verifiers

WMM

GCC

ELVM

x86

Power

ARM

Thank you!
http://www.cs.tau.ac.il/~orilahav/