

# Go's Memory Model

Russ Cox  
rsc@google.com

MIT 6.824 / February 25, 2016

# Why Go?

Go is an answer to problems of scale at Google.

System scale:

- $10^6+$  machines (design point)
- routine to be using 1000+ machines
- coordinating, interacting with lots of other servers
- lots going on at once

Solution: great support for concurrency (channels, goroutines)



# Why Go?

Engineering scale, in 2011:

- 5000+ developers across 40+ offices
- single code tree
- 20+ changes per minute
- 50% code base changes every month
- 50 million test cases executed per day

Solution: design and engineer a language for large code bases  
(simple, understandable, efficient, great tool support)

# Challenges of Go

Concurrency: scalable channels (lock-free?)

Scheduling: efficient use of HW, despite OS barriers?

Polymorphism: efficient interfaces; generics?

Garbage collection: bound pause times, concurrent, per-goroutine?

Program translation: translate other programs to Go?

Race detection: scale to 1000s of goroutines?

**Memory model: what should it be?**

# A Fairy Tale

Once upon a time...

How do you make a program faster?

Wait a year, buy newer hardware.

Is a hardware or compiler optimization valid?

Yes, if valid programs don't change behavior.

(And most programs are valid.)

# An Evil Twist

Hardware engineer's magic spells stopped working.

New magic spell: stamp out more and more cores.

Compiler and operating system engineers give us multithreaded programming.

Now hardware, compiler optimizations change program behaviors.

Those optimizations are invalid.

Or else most programs are invalid.

# Can't we all get along?

“Valid optimizations do not change the behavior of valid programs.”

Which programs are valid? How do we decide?

Is this program valid? Can it ever print 0?

```
// Thread 1
```

```
x = 1;  
done = 1;
```

```
// Thread 2
```

```
while (done == 0) { /* spin */ }  
print(x);
```



# Can this program print 0?

```
// Thread 1
```

```
x = 1;  
done = 1;
```

```
// Thread 2
```

```
while (done == 0) { /* spin */ }  
print(x);
```

“It depends.”

In x86 assembly, no.

In ARM/POWER assembly, yes.

In most C compilers (even on x86), yes (or maybe it won't even finish).

# “It depends” is not a happy ending

Memory (consistency) model defines:

- which programs are valid
- what those valid programs can do
- therefore what programmers can expect
- therefore what compiler writers must ensure / can do

Spoiler alert: memory model happy ending is not here yet.

# This Talk

- Why memory models?
- Hardware
  - Sequential Consistency
  - x86
  - ARM/POWER
  - “Weak ordering” and “data race free”
- Languages / Compilers
  - Java (1996, 2005)
  - C++ (2011, 2014)
  - Go (2009, ...)

# Hardware Memory Models

Now we are writing *assembly language*.

How badly can the hardware mess with us?

# Sequential Consistency

“The customary approach to designing and proving the correctness of multiprocess algorithms for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. A multiprocessor satisfying this condition will be called *sequentially consistent*.”

— Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs” (1979)

# Can this program print 0?

```
// Thread 1
```

```
x = 1;  
done = 1;
```

```
// Thread 2
```

```
while (done == 0) { /* spin */ }  
print(x);
```

# Litmus Test: Message Passing

```
// Thread 1
```

```
x = 1
```

```
y = 1
```

```
// Thread 2
```

```
r1 = y
```

```
r2 = x
```

Can this program see  $r1 = 1, r2 = 0$ ?

# Message Passing w/ Sequential Consistency

x = 1	
y = 1	
	r1 = y
	r2 = x

x = 1	r1 = y
y = 1	r2 = x

x = 1	r1 = y
	r2 = x
y = 1	

	r1 = y
x = 1	
y = 1	
	r2 = x

	r1 = y
x = 1	r2 = x
y = 1	

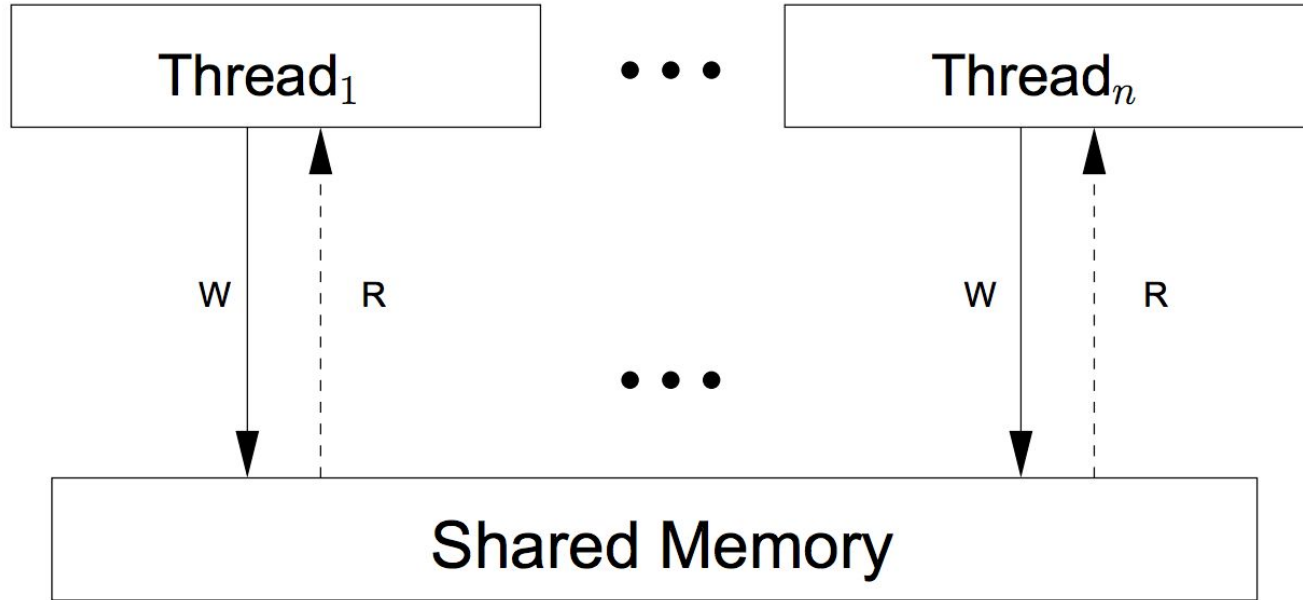
	r1 = y
	r2 = x
x = 1	
y = 1	

Can this program see  $r1 = 1, r2 = 0$ ?

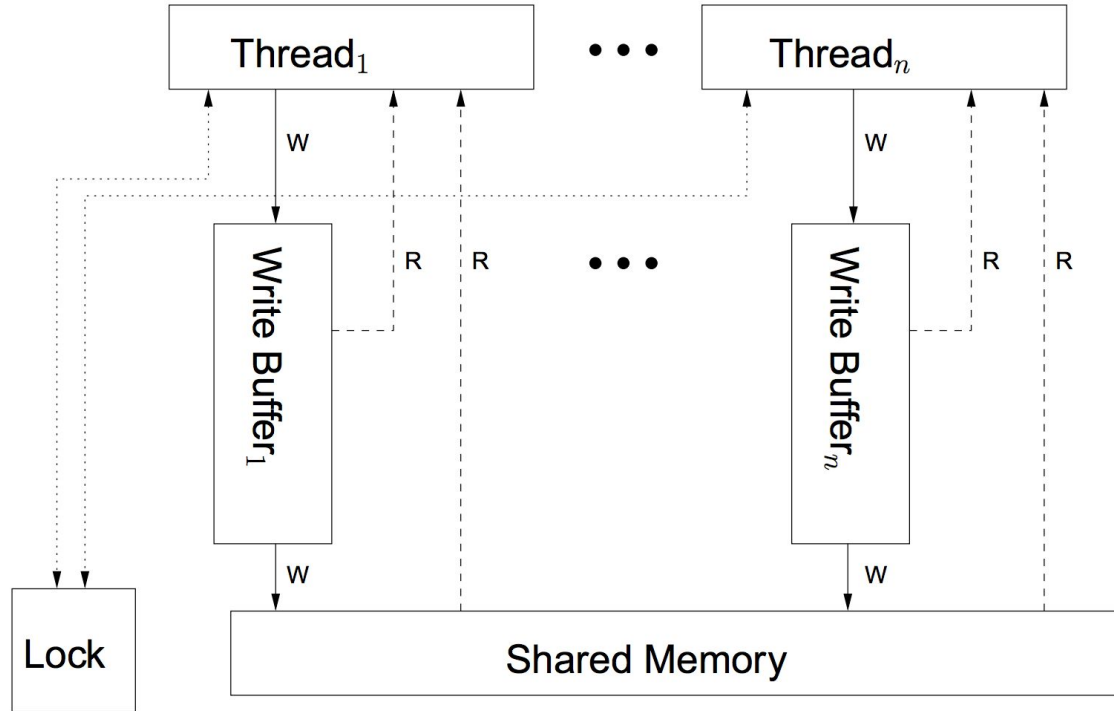
On sequentially consistent hardware: **no**.



# Sequentially Consistent Hardware



# x86 Hardware (Total Store Order)



# Litmus Test: Message Passing

```
// Thread 1
```

```
x = 1
```

```
y = 1
```

```
// Thread 2
```

```
r1 = y
```

```
r2 = x
```

Can this program see  $r1 = 1, r2 = 0$ ?

On x86 (or other TSO): **no**.

Thread 1's writes are observed by other threads in original order.

# Litmus Test: Store Buffering

```
// Thread 1
```

```
x = 1
```

```
r1 = y
```

```
// Thread 2
```

```
y = 1
```

```
r2 = x
```

Can this program see  $r1 = 0, r2 = 0$ ?

On sequentially consistent hw: **no**.

On x86 (or other TSO): **yes!**

Thread 1's local writes are not immediately visible in Thread 2 (and vice versa).

# Memory Fences

```
// Thread 1
```

```
x = 1
```

```
fence
```

```
r1 = y
```

```
// Thread 2
```

```
y = 1
```

```
fence
```

```
r2 = x
```

Can this program see  $r1 = 0$ ,  $r2 = 0$ ? **No.**

Memory fence ensures Thread 1's write is globally visible before Thread 1's read, and vice versa.

Glossing over details.

# Litmus Test: Independent Reads of Indep. Writes

// Thread 1	// Thread 2	// Thread 3	// Thread 4
x = 1	y = 1	r1 = x r2 = y	r3 = y r4 = x

Can this program see  $r1 = 1, r2 = 0, r3 = 1, r4 = 0$ ?

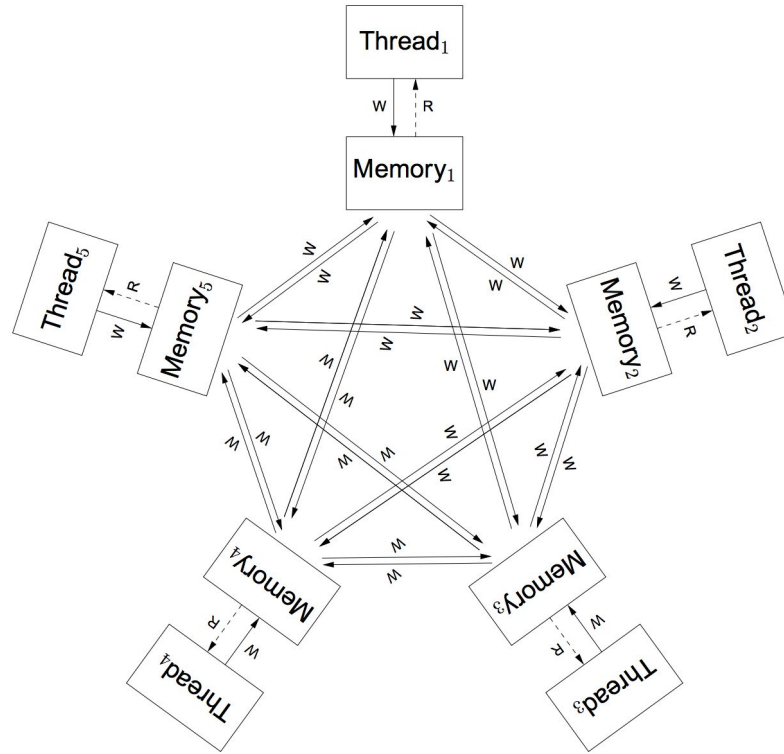
Can Thread 3 see x change before y but Thread 4 see the opposite?

On sequentially consistent hw: **no**.

On x86 (or other TSO): **no**.

There is a total order over all stores to main memory.

# ARM/POWER Hardware



# Litmus Test: Message Passing

```
// Thread 1
```

```
x = 1
```

```
y = 1
```

```
// Thread 2
```

```
r1 = y
```

```
r2 = x
```

Can this program see  $r1 = 1, r2 = 0$ ?

On x86 (or other TSO): **no**.

On ARM/POWER: **yes!**

Thread 1's writes **may not be** observed by other threads in original order.



# Litmus Test: Store Buffering

```
// Thread 1
```

```
x = 1
```

```
r1 = y
```

```
// Thread 2
```

```
y = 1
```

```
r2 = x
```

Can this program see  $r1 = 0, r2 = 0$ ?

On sequentially consistent hw: **no**.

On x86 (or other TSO): **yes!**

On ARM/POWER: **yes!**

Thread 1's local writes are not immediately visible in Thread 2 (and vice versa).

# Litmus Test: Independent Reads of Indep. Writes

// Thread 1	// Thread 2	// Thread 3	// Thread 4
x = 1	y = 1	r1 = x r2 = y	r3 = y r4 = x

Can this program see  $r1 = 1, r2 = 0, r3 = 1, r4 = 0$ ?

(Can Thread 3 see x change before y but Thread 4 see the opposite?)

On sequentially consistent hw: **no**.

On x86 (or other TSO): **no**.

On ARM/POWER: **yes!**

Different threads may receive different writes in different orders.

# Litmus Test: Coherence

// Thread 1	// Thread 2	// Thread 3	// Thread 4
x = 1	x = 2	r1 = x r2 = x	r3 = x r4 = x

Can this program see  $r1 = 1, r2 = 2, r3 = 2, r4 = 1$ ?

(Can Thread 3 see  $x=1$  before  $x=2$  but Thread 4 see the opposite?)

On sequentially consistent hw: **no**.

On x86 (or other TSO): **no**.

On ARM/POWER: **no**.

Threads must agree which writes overwrite other writes.

# Weak Ordering

“Let a *synchronization model* be a set of constraints on memory accesses that specify how and when synchronization needs to be done.

Hardware is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obey the synchronization model.”

— Adve and Hill, “Weak Ordering - A New Definition” (1990)

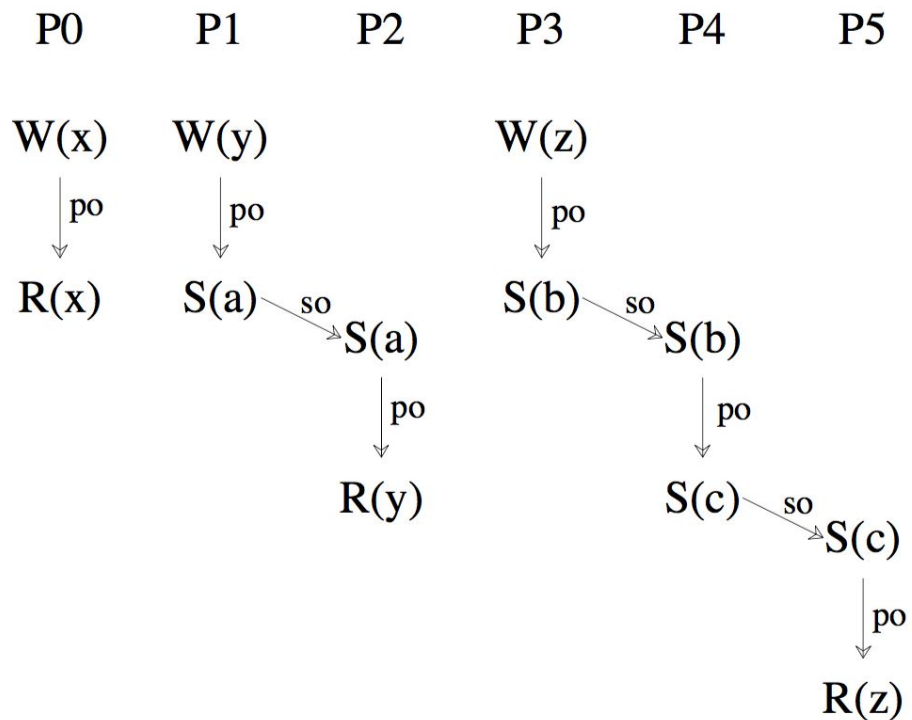
# Data-Race-Free (DRF)

Synchronization operations are (to hardware) recognizably different from ordinary operations.

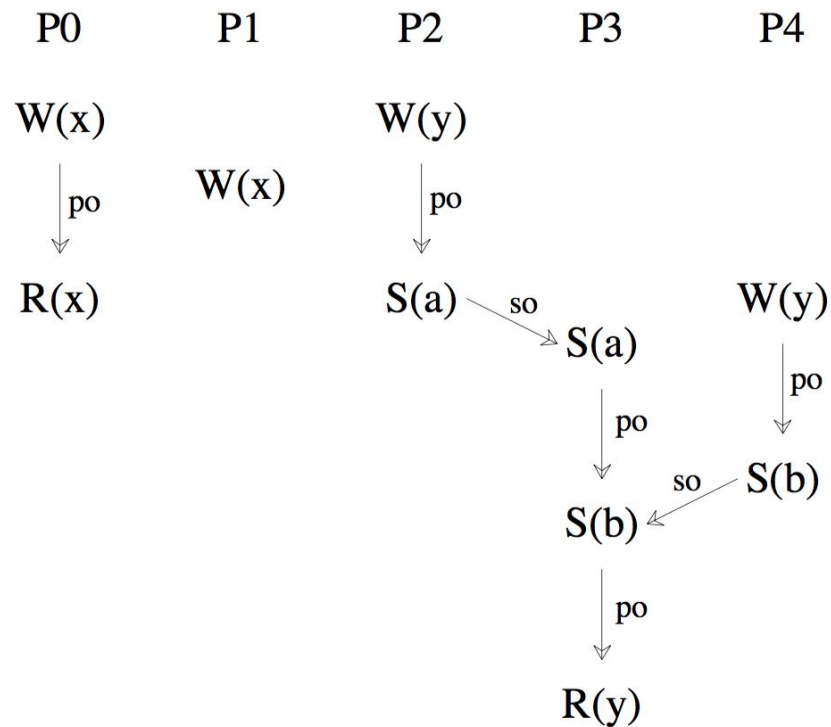
A program is data-race-free if for all idealized SC executions, any two ordinary memory accesses to the same location from different threads are either:

- both reads
- or separated by synchronization operations:  
*one happens before the other*

# Data-Race-Free



# Not Data-Race-Free



# Hardware weakly ordered by DRF

“Hardware is weakly ordered with respect to [DRF] if and only if it appears sequentially consistent to all software that obey [DRF].”



# Hardware weakly ordered by DRF when...

- Intra-processor dependencies are preserved
- All writes to same location have a global total order (coherence)
- All sync operations to same location have a global total order
  - for S1 before S2, all of S1 must complete before any of S2 starts.
- A new access is not generated by a processor until previous sync operations are committed.
- Once a sync operation S by processor P is committed, no other sync operations on the same location can commit until:
  - all reads by P before S must be committed
  - all writes by P before S must be globally performed

# Hardware weakly ordered by DRF

“Hardware is weakly ordered with respect to [DRF] if and only if it appears sequentially consistent to all software that obey [DRF].”

Basically everything, given appropriate synchronization implementations

- Earlier hardware models
- VAX
- x86
- ARM/POWER

Guarantee that DRF implies appearance of SC: DRF-SC.

# Compilers

Significant freedom to rewrite code

Significant gaps in knowledge of execution

w = 1

x = 2

r1 = y

r2 = z

Compiled code fails (answers **yes!** to) every litmus test we've seen, including coherence!

# Compiler Optimizations

Is this a valid optimization?

```
if(c) {  
    x++  
} else {  
    ... lots of code ...  
}
```

```
x++  
if(!c) {  
    x--  
    ... lots of code ...  
}
```

Compiler and language must be involved in multithreaded guarantees.

Boehm, “Threads Cannot be Implemented as a Library” (2004)

# Weak Ordering?

“**Hardware** is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obey the synchronization model.”

Why not a programming language (implementation)?

# Original Java Memory Model (1996)

Defined “volatile” for atomic synchronization

“Coherence” for ordinary variables

Specified behavior of programs with data races

Very complex, difficult to interpret, too weak, too strong

# Original Java: Too Weak

```
int x;  
volatile boolean done;
```

```
// Thread 1  
x = 1;  
done = true;
```

```
// Thread 2  
while(!done) { }  
print(x)
```

Volatiles do not impose ordering constraints on non-volatiles!

Compiler might reorder the writes, not to mention hardware.

# Original Java: Too Strong

```
// Thread 1  
int i = x.f;  
  
int j = y.f;  
int k = x.f; // (= i?)
```

```
// Thread 2  
  
x.f++;
```

Compiler wants to use “k = i”

But if  $x == y$  (same pointer), by coherence,  
k cannot load old value after j loads new value.

Compiler cannot do common subexpression elimination!

Pugh, “Fixing the Java Memory Model” (1999)



# New Java Memory Model (2005)

DRF-SC: Data-Race-Free programs run like a Sequentially Consistent execution.

Must define base cases for happens-before:

- Unlock of mutex  $m$  happens before subsequent lock of  $m$
- Write to volatile variable  $v$  happens before subsequent read of  $v$
- Creation of thread happens before first action in thread
- ...

Definition of subsequent: according to some underlying total ordering.

Defined semantics even for programs with data races.

# Litmus Test: Store Buffering

```
// Thread 1
```

```
x = 1  
r1 = y
```

```
// Thread 2
```

```
y = 1  
r2 = x
```

Can this program see  $r1 = 0, r2 = 0$ ?

On sequentially consistent hw: **no**.

On x86 (or other TSO): **yes!**

On Java (using volatiles): **no**.

Volatile results must be as in some total ordering (both stores and loads).

So we need fences on x86 (and others).

# New Java Memory Model (2005)

Define semantics for racy programs, to...

- support Java's security and safety guarantees
- make it easier to track down errors
- make it harder for attackers to exploit errors
- make it clearer to programmers what their programs do

# New Java Memory Model (2005)

- Each read of a variable  $x$  sees a write to  $x$ .
- Execution obeys happens-before consistency:
  - If a read  $r$  observes  $w$ ,  
     $r$  cannot happen before  $w$ .
  - If a read  $r$  observes  $w$ ,  
    there is no  $w'$  such that  $w$  happens before  $w'$  happens before  $r$ .
- ...

# New Java Memory Model (2005)

Major improvement over original model.

- More guarantees for programmer.

- More compiler optimizations definitively allowed.

Still the memory model for Java.

Still not quite “right.”

# Happens-Before

Happens-before was defined to specify **whether a program has a race**.

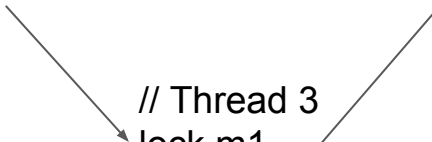
Can it be used to specify the **possible behaviors** of a program? Not quite.

# Incoherence of Happens-Before

```
// Thread 1  
lock m1  
x = 1  
unlock m1
```

```
// Thread 2  
lock m2  
x = 2  
unlock m2
```

```
// Thread 3  
lock m1  
lock m2  
r1 = x  
r2 = x  
unlock m2  
unlock m1
```



“r1 = x” can see either write.

“r2 = x” can see either write.

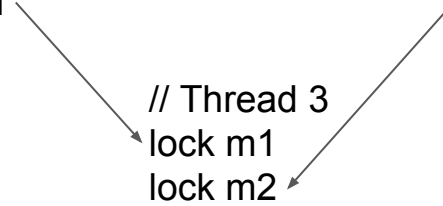
Can they disagree? **Yes!**

# Incoherence of Happens-Before

```
// Thread 1
lock m1
x = 1
unlock m1

// Thread 2
lock m2
x = 2
unlock m2

// Thread 3
lock m1
lock m2
r1 = x
x = r1
r2 = x
unlock m2
unlock m1
```

The diagram shows three threads. Thread 1 locks m1 and sets x=1. Thread 2 locks m2 and sets x=2. Thread 3 locks both m1 and m2, reads x into r1, updates x with r1 (highlighted in blue), reads x into r2, and then unlocks both m2 and m1. Arrows point from the 'unlock m1' of Thread 1 and 'unlock m2' of Thread 2 to the 'lock m1' and 'lock m2' of Thread 3.

“r1 = x” can see either write.

“r2 = x” must see preceding write.

Can they disagree? **No.**



# Incomplete JMM => Invalid Compiler Optimizations

By dropping the second instruction in:

```
r1 = x
```

```
x = r1
```

The compiler changes a program that **cannot** see different r1, r2 into one that **can** see different r1, r2.

Therefore this optimization is invalid according to the Java Memory Model.

Surprise!

Ševčík and Aspinall, “On the Validity of Program Transformations in the Java Memory Model” (2007)

# Acausality of Happens-Before

```
// Thread 1 (y = x)
```

```
r1 = x  
y = r1
```

```
// Thread 2 (x = y)
```

```
r2 = y  
x = r2
```

Happens-before does not exclude  $x = y = r1 = r2 = 42$  (!)

Each read can see a racing write, but that forms a causality cycle.

“Out of thin air values.”

Program is racy, but still don't want 42.

# Acausality of Happens-Before

```
// Thread 1 (if (x==1) y=1)
```

```
r1 = x  
if (r1 == 1)  
    y = 1
```

```
// Thread 2 (if (y==1) x=1)
```

```
r2 = y  
if (r2 == 1)  
    x = 1
```

Happens-before does not exclude  $x = y = r1 = r2 = 1$  (!)

Each read can see a **dead** racing write, but that forms a causality cycle.

“Out of thin air values.”

Program is **not** racy: by DRF-SC must only show  $x = y = r1 = r2 = 0$ .

# Acausality of Happens-Before

Happens-before by itself is insufficient to define program semantics:  
Java Memory Model (2005) adds separate rules to define valid executions.

# Acausality of Happens-Before

Happens-before by itself is insufficient to define program semantics:  
Java Memory Model (2005) adds separate rules to define valid executions.

“Prohibiting such causality violations in a way that does not also prohibit other desired optimizations turned out to be surprisingly difficult. ... After many proposals and five years of spirited debate, the current model was approved as the best compromise. ... Unfortunately, this model is very complex, was known to have some surprising behaviors, and has recently been shown to have a bug.”

— Adve and Boehm, “Memory Models: A Case for Rethinking Parallel Languages and Hardware” (2010)

# C++11 Memory Model

Inspired by Java, C++0x (later C++11) added memory model.

Data-Race-Free implies Sequentially-Consistent (DRF-SC)

But programs with data races completely undefined!

Atomic variables like Java's volatile

But also some weaker atomics.

# C++11 Atomics

`memory_order_seq_cst` (like Java volatile):

A read that observes a write creates a happens-before edge.  
An underlying total order over all operations dictates  
which reads see which writes.

`memory_order_acq`, `memory_order_rel`, `memory_order_acq_rel`:

A read that observes a write creates a happens-before edge.  
**No underlying total order: reads can miss writes.**

# Litmus Test: Store Buffering

```
// Thread 1
```

```
x = 1  
r1 = y
```

```
// Thread 2
```

```
y = 1  
r2 = x
```

Can this program see  $r1 = 0, r2 = 0$ ?

On sequentially consistent hw: **no**.

On x86 (or other TSO): **yes!**

On Java (using volatiles): **no**.

On C++11 (memory\_order\_seq\_cst): **no**.

On C++11 (memory\_order\_acq\_rel): **yes!**



# C++11 Atomics

`memory_order_relaxed` (like ordinary Java variables):

- No happens-before edges.

- Must define what can be observed.

- Races here aren't races for DRF.

`memory_order_consume`:

- Something about data/control dependencies.

- “The one no one understands.”

# C++11 “DRF-SC or Catch Fire”

Data-Race-Free programs execute as if Sequentially Consistent.

But programs with data races can misbehave or crash in arbitrarily exotic ways!

- Pthreads and win32 made no guarantees either.
- Existing compilers too hard to fix.
- C++ object initialization would be get more expensive.
- Programmers who need racy programs can use `memory_order_relaxed`.
- Leaving race semantics undefined makes race detector legal.
- Disallowing races eliminates false positives in race detector.

Boehm, “Memory Model Rationales” (2007)

Boehm and Adve, “Foundations of the C++ Concurrency Memory Model” (2008)

# C++11 “DRF-SC or Catch Fire”

Non-reason to disallow data races: simpler.

Still have all the causality problems, for `memory_order_relaxed` atomics!

# C++11: Another Attempt At Causality

```
// Thread 1 (if (x==1) y=1)
```

```
r1 = x  
if (r1 == 1)  
    y = 1
```

```
// Thread 2 (if (y==1) x=1)
```

```
r2 = y  
if (r2 == 1)  
    x = 1
```

C++11 disallows  $r1 = r2 = x = y = 1$  for ordinary variables (DRF-SC) but somehow allows it for `memory_model_relaxed` atomics!

Vafeiadis et al, “Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it” (2015)

# C++11: Another Attempt At Causality

C++11: A few rules to disallow certain kinds of out-of-thin-air values  
+ informal advice to discourage other kinds.

The set of rules turned out to be “both insufficient, in that it leaves it largely impossible to reason about programs with `memory_order_relaxed`, and seriously harmful, in that it arguably disallows all reasonable implementations of `memory_order_relaxed` on architectures like ARM and POWER.”

C++14: Informal advice only.

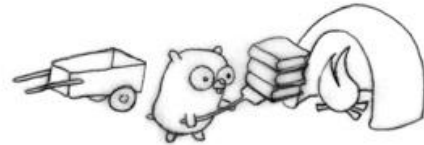
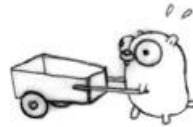
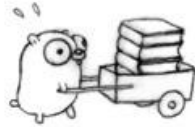
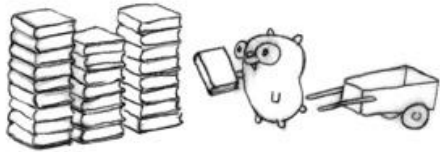
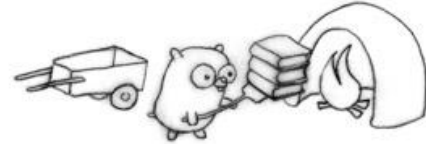
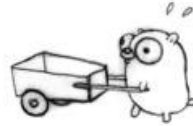
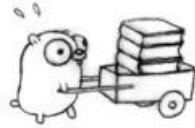
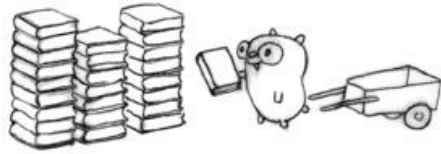
Boehm, “Prohibiting “out of thin air” results in C++14” (2013)

# Causality is Hard

Batty et al, “The Problem of Programming Language Concurrency Semantics” (2015):

“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.”

# Go Memory Model?



# Go Memory Model

Two purposes:

- Make guarantees for programmers.
- Allow compilers/hardware to make certain changes to programs.

Ideally, perfectly balanced. In practice, more conservative: might do neither.

Explicit concern: Leave room for future refinement, refraining from:

- making debatable guarantees to programmers
- allowing compilers/hardware to make debatable changes to programs



# Go Memory Model

Advice:

“Programs that modify data being simultaneously accessed by multiple goroutines must serialize such access. To serialize access, protect the data with channel operations or other synchronization primitives such as those in the sync and sync/atomic packages.

If you must read the rest of this document to understand the behavior of your program, you are being too clever.

**Don't be clever.”**

# Go Memory Model

Semantics based on happens-before:

- If p imports q, q's init happens before p's.
- Package main's init happens before main.main
- The go statement happens before the created goroutine's execution
- A send (or close) on a channel happens before the receive
- Unlock happens before subsequent Lock

# Go Memory Model

A read  $r$  is allowed to observe a write  $w$  to location  $v$  if:

- $r$  does not happen before  $w$
- There is no write  $w'$  to  $v$  such that  $w < w' < r$ . [ $<$  is happens-before]

A read  $r$  is therefore guaranteed to observe  $w$  if:

- $w$  happens before  $r$  ( $w < r$ )
- For any other write  $w'$  to  $v$ ,  $w' < w$  or  $r < w'$

Intent: like DRF-SC without allowing compilers to ruin programmers' lives.

# Go Memory Model

Arguments in favor of semantics for programs with data races for simple data:

- Limit damage caused by (we hope accidental) introduction of races.
- How do you debug a program if the bug causes undefined behavior?
- Do large multithreaded programs without data races even exist?
- Concurrent garbage collector is one big racing read.

Races on interface, string, slice, map entries can still do arbitrarily bad things :-)

# Go Atomic Operations

Package `sync/atomic` is conspicuously missing.

Proposal: match Java `volatile` and C++ `memory_order_seq_cst`.

- An atomic write happens before an atomic read that observes the write.
- The outcome of atomic operations must be consistent with a total order over all atomic operations in the program.

# Go Causality

No text about causality at all.

I didn't think it was worth saying.

Should probably add informal exclusion a la C++14.

Clearly not worth trying to formalize!

Questions?