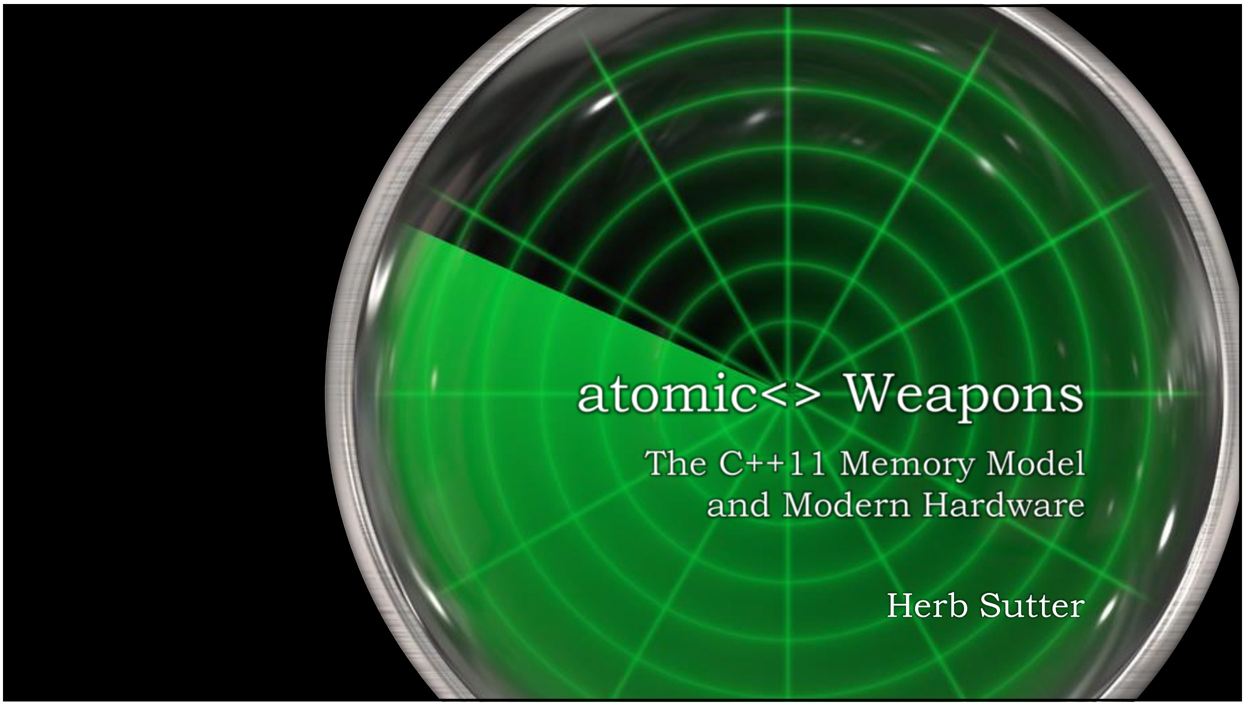


atomic<> Weapons

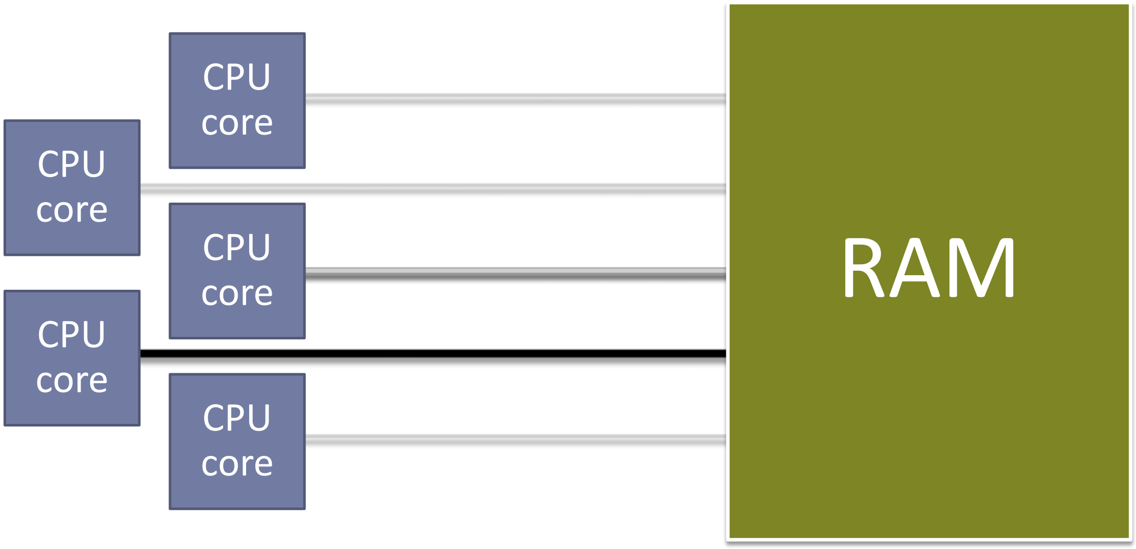
The C++11 Memory Model and Modern Hardware

Herb Sutter

Roadmap

- Optimizations, Races, and the Memory Model
- Ordering – What: Acquire and Release
- Ordering – How: Mutexes, Atomics, and/or Fences
- Other Restrictions on Compilers and Hardware (→ Bugs)
- Code Gen & Performance: x86/x64, IA64, POWER, ARM, ... ???
- Relaxed Atomics (as time allows)
- Coda: Volatile (as time allows)

The Machine Everyone Codes For



CPU core

CPU core

CPU core

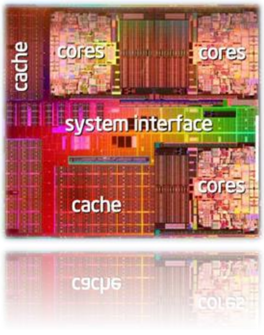
CPU core

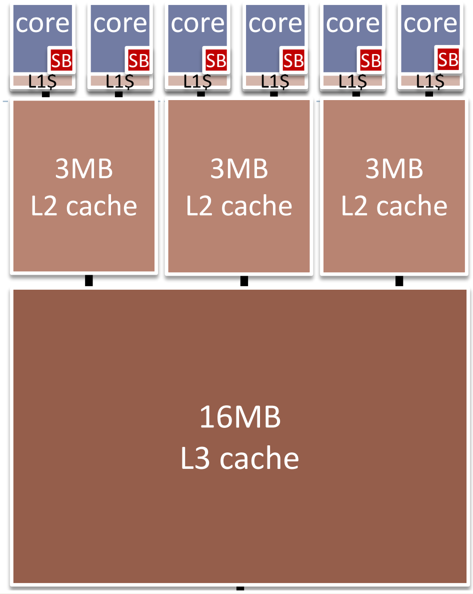
CPU core

CPU core

RAM

2008 Real HW: Intel Dunnington





core SB L1$

core SB L1$

core SB L1$

core SB L1$

core SB L1$

core SB L1$

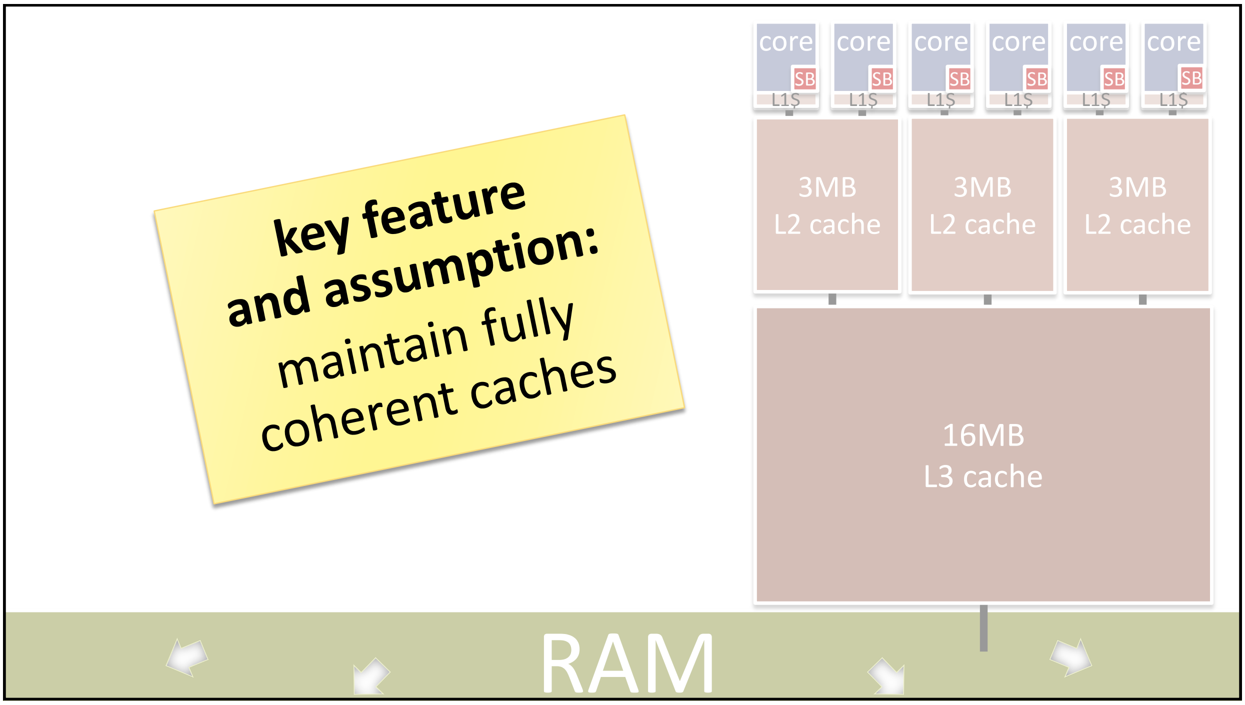
3MB L2 cache

3MB L2 cache

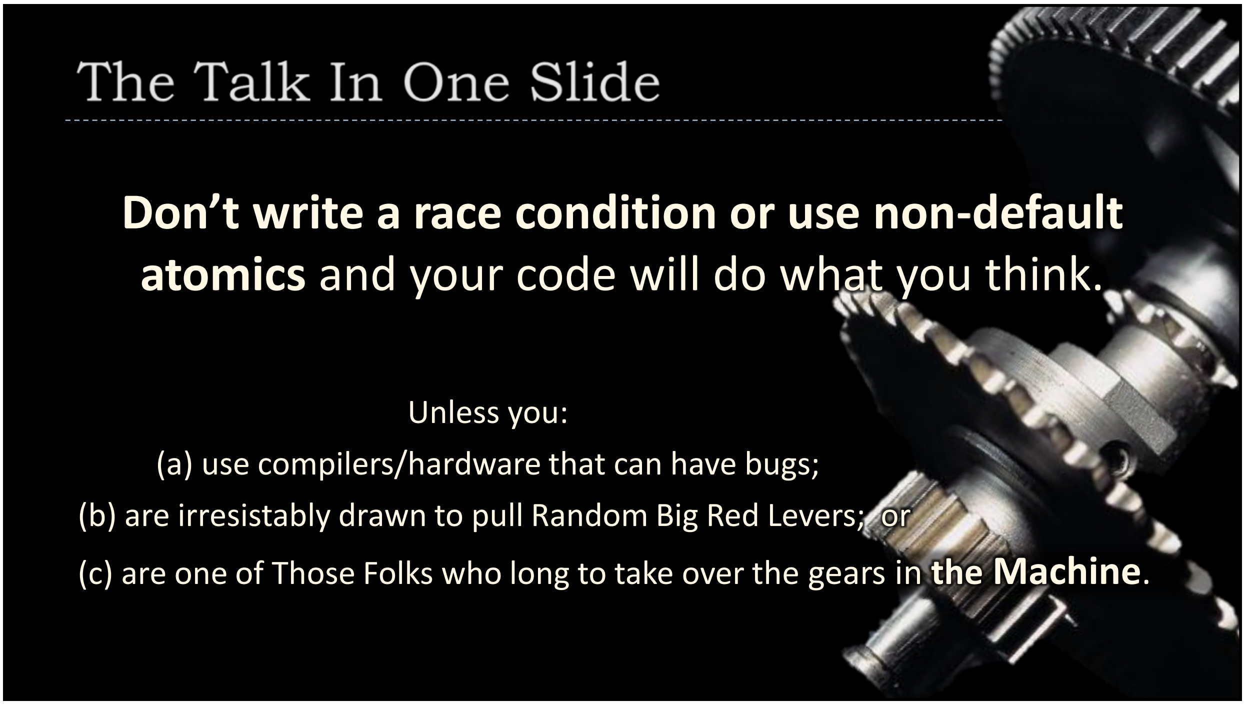
3MB L2 cache

16MB L3 cache

RAM



key feature and assumption:
maintain fully coherent caches



The Talk In One Slide

Don't write a race condition or use non-default atomics and your code will do what you think.

Unless you:

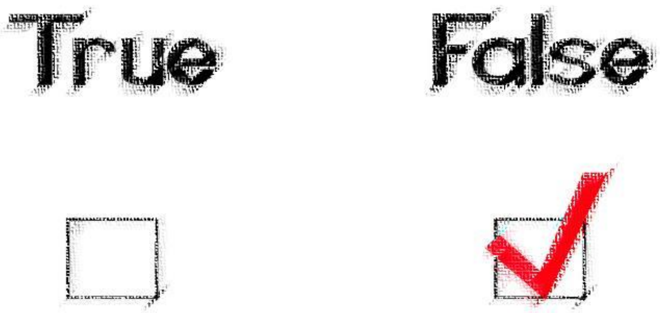
(a) use compilers/hardware that can have bugs;

(b) are irresistably drawn to pull Random Big Red Levers; or

(c) are one of Those Folks who long to take over the gears in **the Machine**.

The Truth

- Q: Does your computer execute the program you wrote?



The Truth

- Q: Does your computer execute the program you wrote?
- A: What a quaint concept!
  - On big iron, contemporary with live Beatles performances.
  - On PCs, contemporary with leg warmers.
- *Think: Compiler optimization, processor OoO execution, cache coherency.*

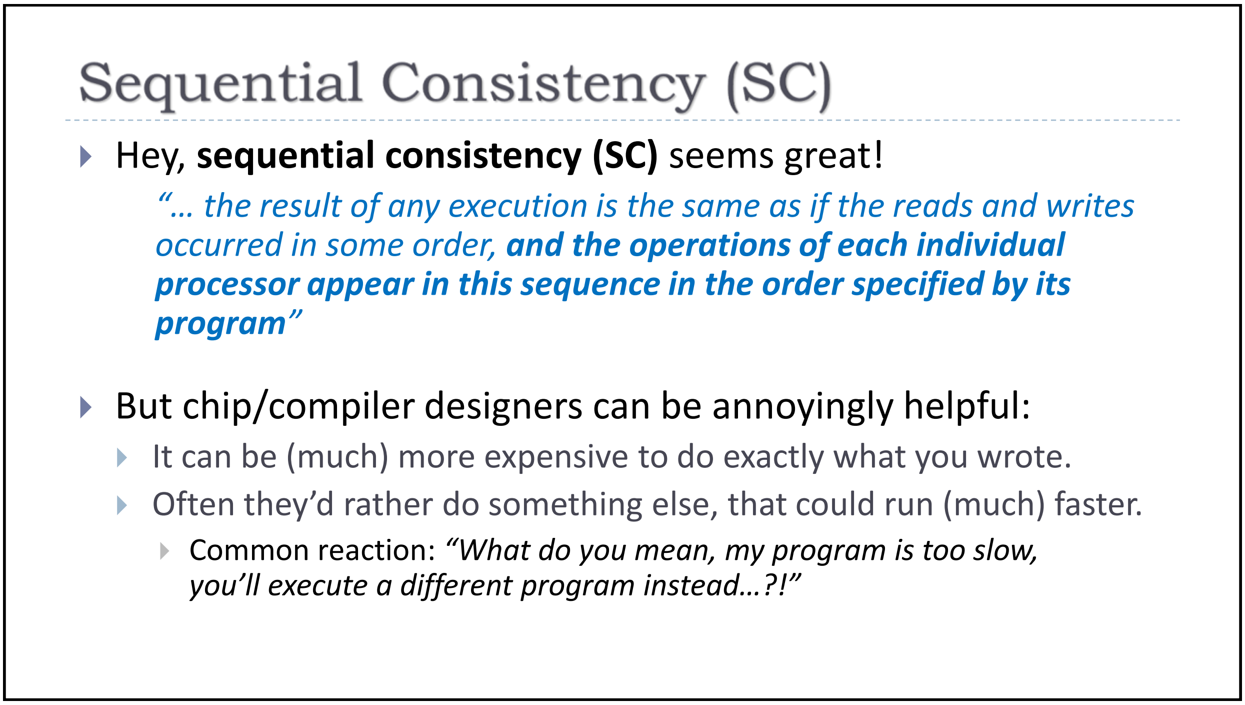
The Truth

- Compiler/processor/cache says:

> "No, it's much better to **execute a different program**.
>
> Hey, don't complain. It's for your own good. You really wouldn't want to execute that *dreck* you actually wrote."

Two Key Concepts

- **Sequential consistency (SC)**: Executing the program you wrote.
  - Defined in 1979 by Leslie Lamport as *"the result of any execution is the same as if the reads and writes occurred in some order, **and the operations of each individual processor appear in this sequence in the order specified by its program**"*
- **Race condition**: A memory location (variable) can be simultaneously accessed by two threads, and at least one thread is a writer.
  - Memory location == non-bitfield variable, or sequence of non-zero-length bitfield variables.
  - Simultaneously == without happens-before ordering.

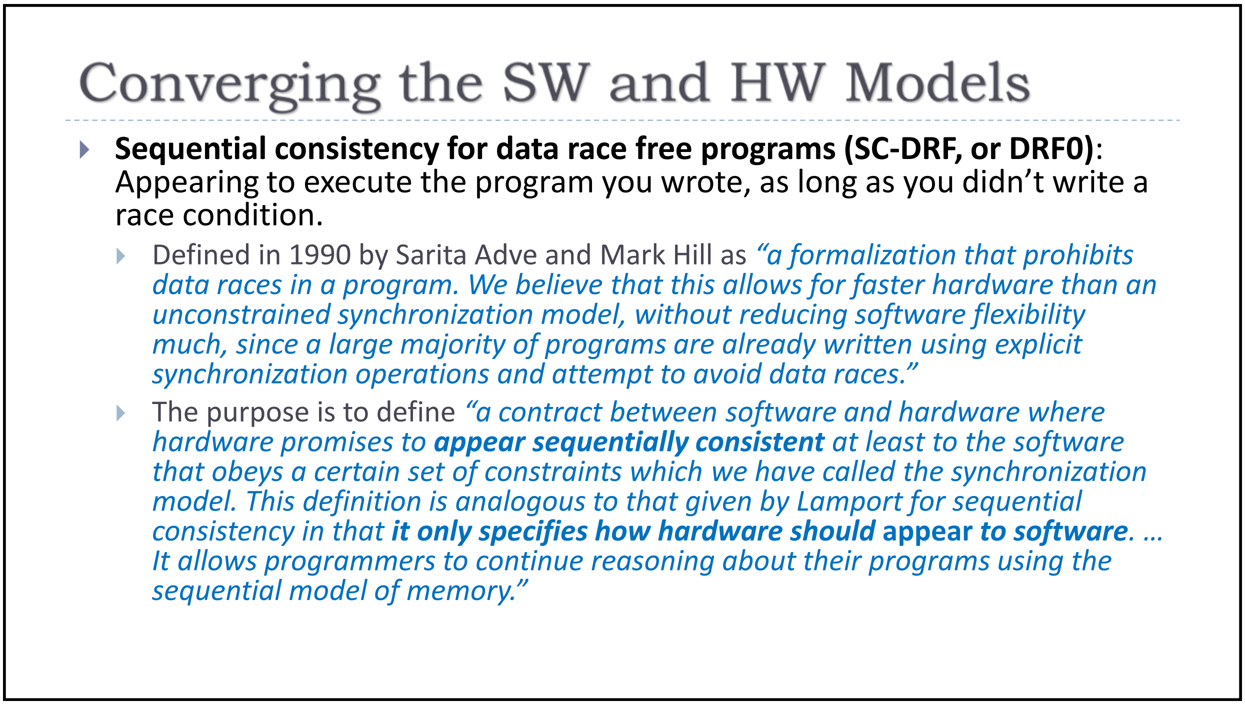


Sequential Consistency (SC)

- Hey, **sequential consistency (SC)** seems great!

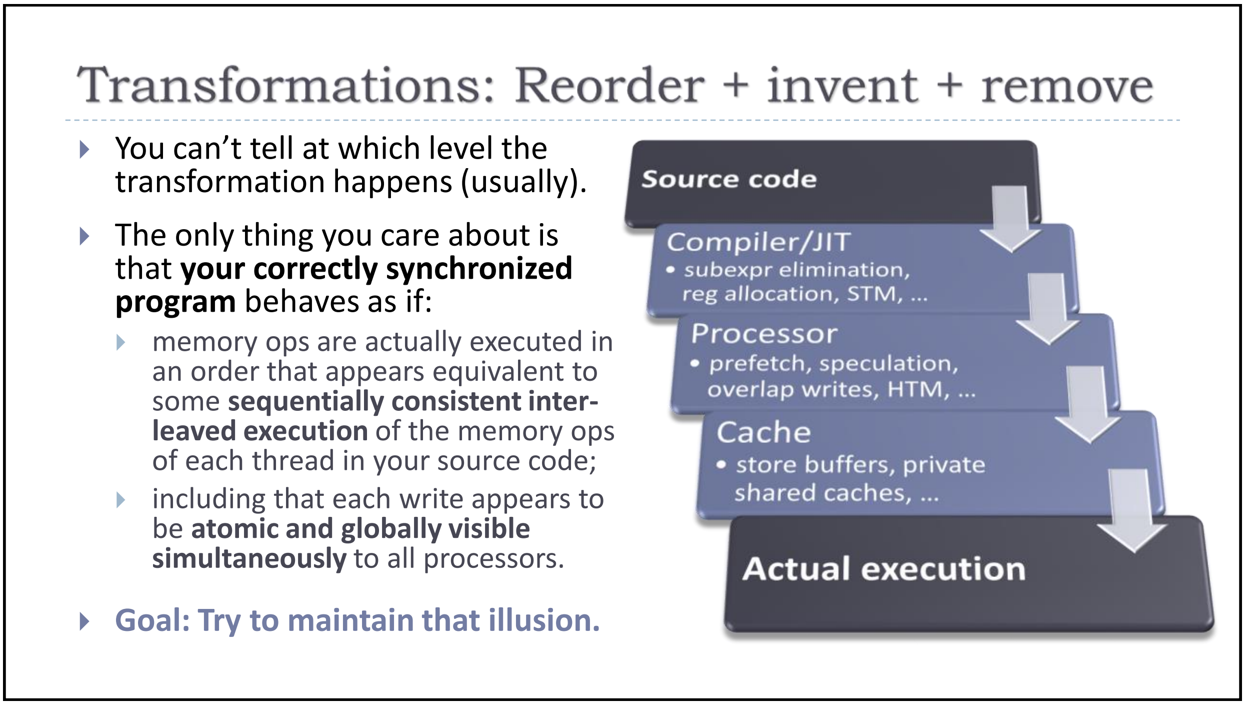
  *"… the result of any execution is the same as if the reads and writes occurred in some order, **and the operations of each individual processor appear in this sequence in the order specified by its program**"*

- But chip/compiler designers can be annoyingly helpful:
  - It can be (much) more expensive to do exactly what you wrote.
  - Often they'd rather do something else, that could run (much) faster.
    - Common reaction: *"What do you mean, my program is too slow, you'll execute a different program instead…?!"*



Converging the SW and HW Models

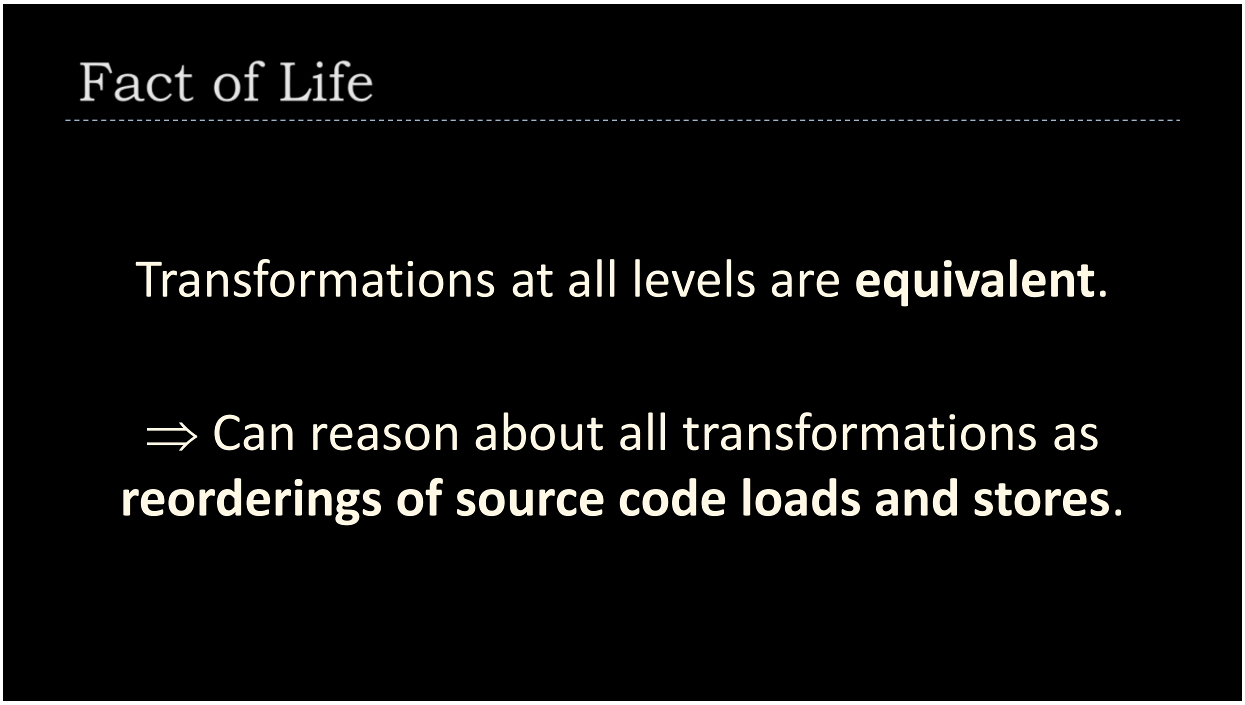
- **Sequential consistency for data race free programs (SC-DRF, or DRF0)**: Appearing to execute the program you wrote, as long as you didn't write a race condition.
  - Defined in 1990 by Sarita Adve and Mark Hill as *"a formalization that prohibits data races in a program. We believe that this allows for faster hardware than an unconstrained synchronization model, without reducing software flexibility much, since a large majority of programs are already written using explicit synchronization operations and attempt to avoid data races."*
  - The purpose is to define *"a contract between software and hardware where hardware promises to **appear sequentially consistent** at least to the software that obeys a certain set of constraints which we have called the synchronization model. This definition is analogous to that given by Lamport for sequential consistency in that **it only specifies how hardware should appear to software**. … It allows programmers to continue reasoning about their programs using the sequential model of memory."*



Transformations: Reorder + invent + remove

- You can't tell at which level the transformation happens (usually).
- The only thing you care about is that **your correctly synchronized program** behaves as if:
  - memory ops are actually executed in an order that appears equivalent to some **sequentially consistent interleaved execution** of the memory ops of each thread in your source code;
  - including that each write appears to be **atomic and globally visible simultaneously** to all processors.
- **Goal: Try to maintain that illusion.**

Source code
Compiler/JIT
• subexpr elimination, reg allocation, STM, …
Processor
• prefetch, speculation, overlap writes, HTM, …
Cache
• store buffers, private shared caches, …
Actual execution



Fact of Life

Transformations at all levels are **equivalent**.

⇒ Can reason about all transformations as **reorderings of source code loads and stores.**

Dekker's and Peterson's Algorithms

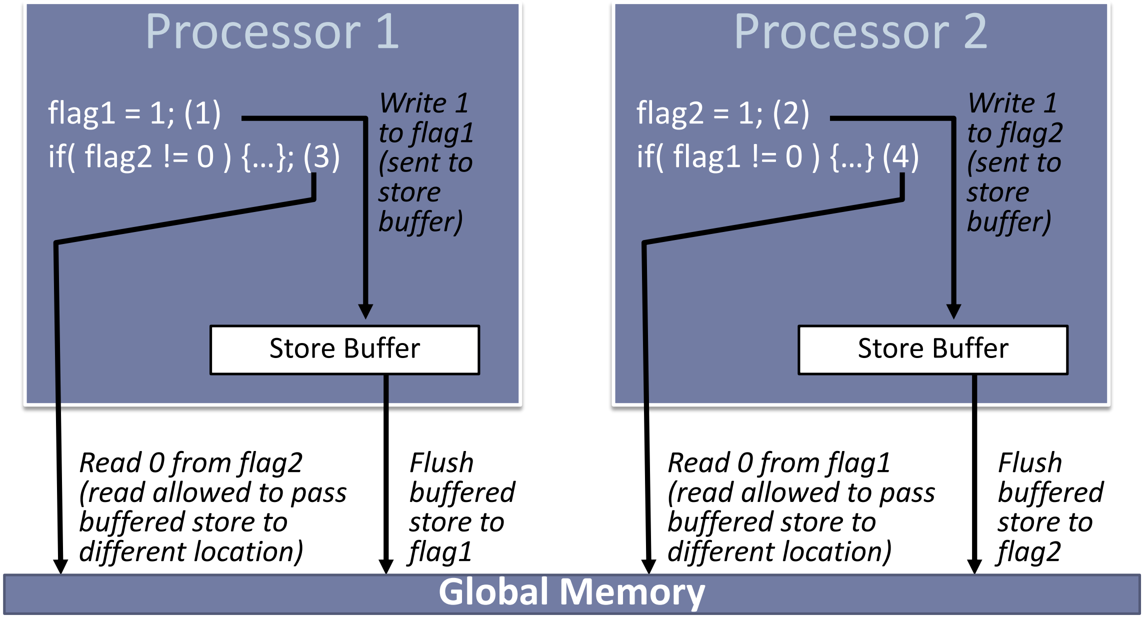
- Consider (flags are shared and atomic *but unordered*, initially zero):
  - Thread 1:

```
flag1 = 1;          // a: declare intent
if( flag2 != 0 )    // b
   // resolve contention
else
   // enter critical section
```

  - Thread 2:

```
flag2 = 1;          // c: declare intent
if( flag1 != 0 )    // d
   // resolve contention
else
   // enter critical section
```

- Q: Could both threads enter the critical region?
  - **Maybe:** If a can pass b, or c can pass d, this breaks.
  - Solution 1 (good): Use a suitable atomic type (e.g., Java/≈.NET "volatile", C++11 **std::atomic<>**) for the flag variables.
  - Solution 2 (good?): Use system locks instead of rolling your own.
  - Solution 3 (problematic): Write a memory barrier after a and c.



Processor 1

flag1 = 1; (1)

if(flag2 != 0) {…}; (3)

Write 1 to flag1 (sent to store buffer)

Store Buffer

Read 0 from flag2 (read allowed to pass buffered store to different location)

Flush buffered store to flag1

Processor 2

flag2 = 1; (2)

if(flag1 != 0) {…} (4)

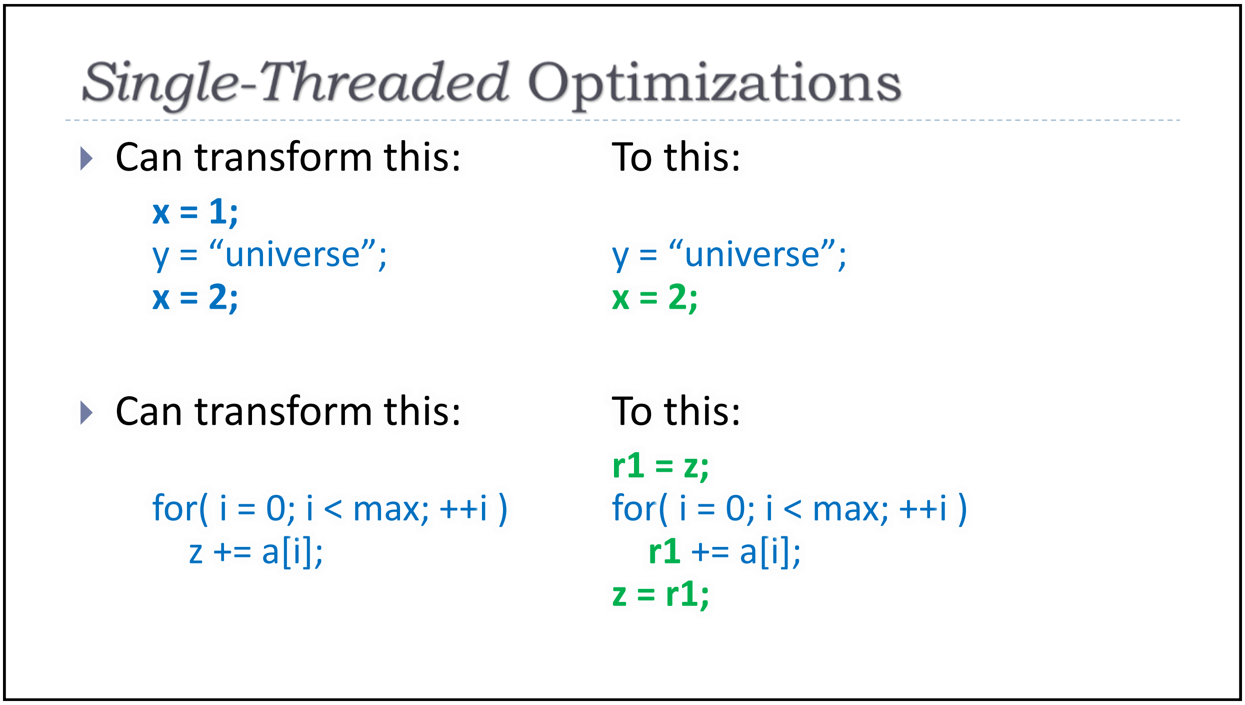
Write 1 to flag2 (sent to store buffer)

Store Buffer

Read 0 from flag1 (read allowed to pass buffered store to different location)

Flush buffered store to flag2

Global Memory



Single-Threaded Optimizations

- Can transform this:

```
x = 1;
y = “universe”;
x = 2;
```

To this:

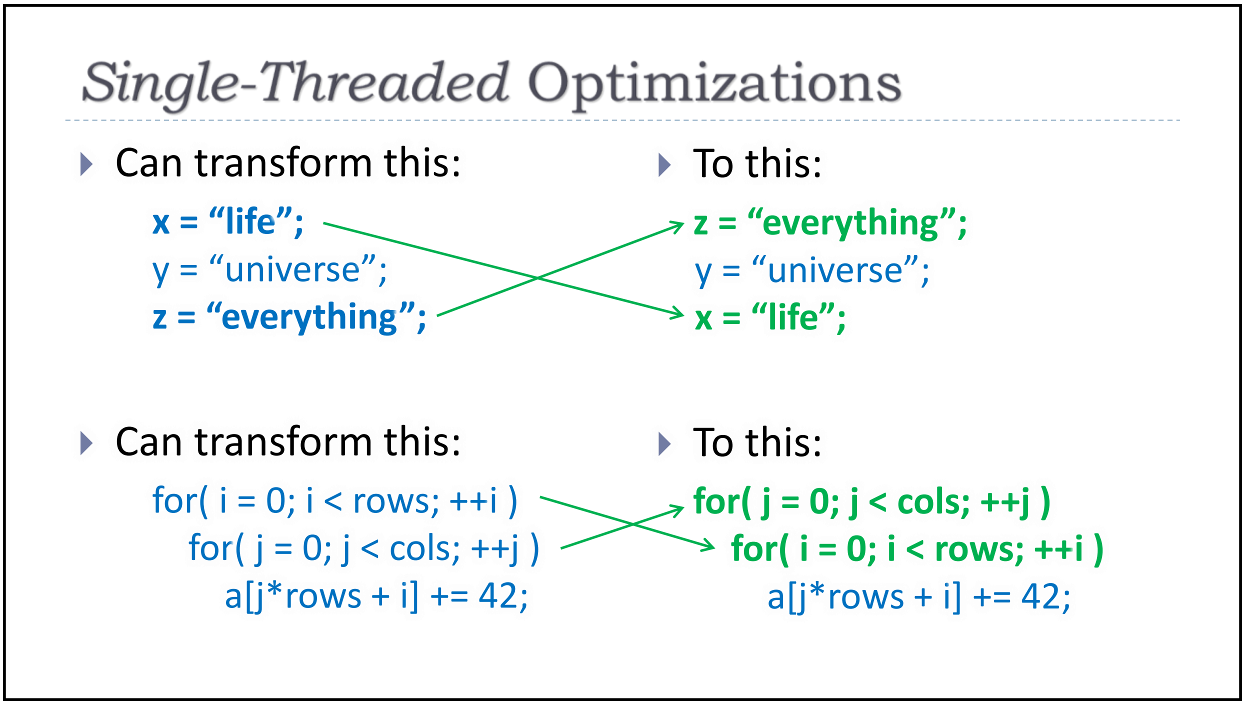
```
y = “universe”;
x = 2;
```

- Can transform this:

```
for( i = 0; i < max; ++i )
   z += a[i];
```

To this:

```
r1 = z;
for( i = 0; i < max; ++i )
   r1 += a[i];
z = r1;
```



Single-Threaded Optimizations

- Can transform this:

```
x = “life”;
y = “universe”;
z = “everything”;
```

- To this:

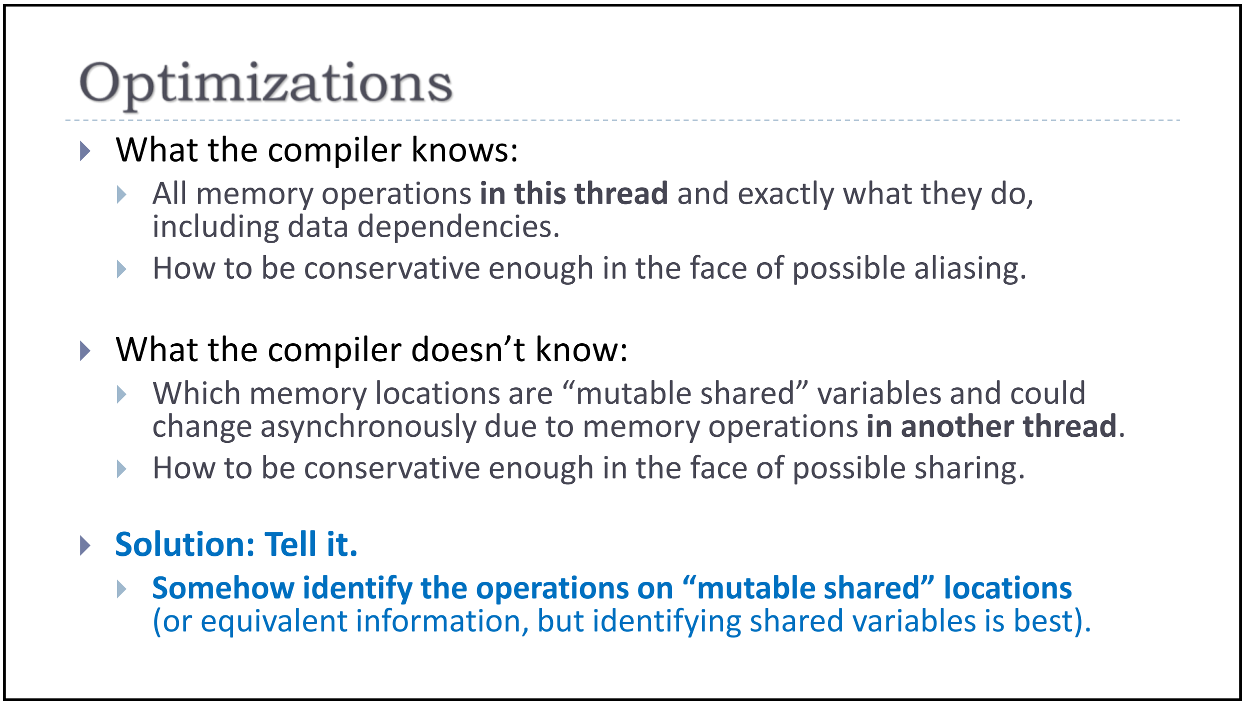
```
z = “everything”;
y = “universe”;
x = “life”;
```

- Can transform this:

```
for( i = 0; i < rows; ++i )
   for( j = 0; j < cols; ++j )
      a[j*rows + i] += 42;
```

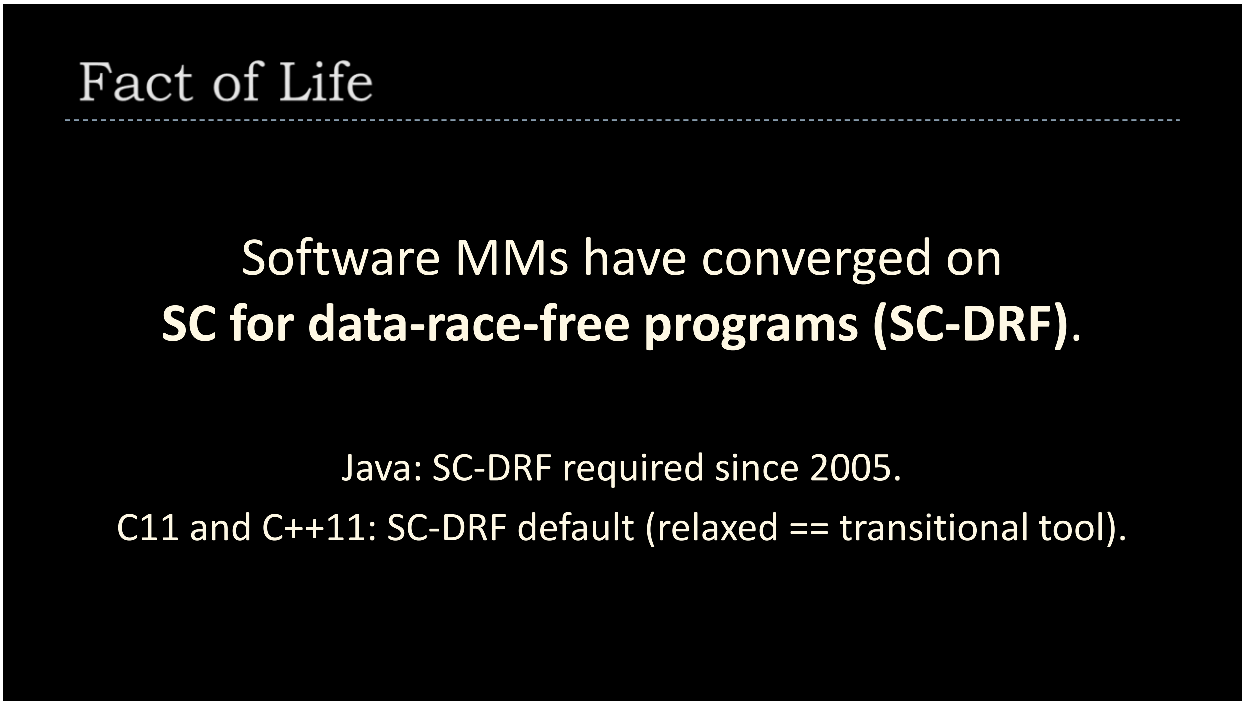
- To this:

```
for( j = 0; j < cols; ++j )
   for( i = 0; i < rows; ++i )
      a[j*rows + i] += 42;
```



Optimizations

- What the compiler knows:
  - All memory operations **in this thread** and exactly what they do, including data dependencies.
  - How to be conservative enough in the face of possible aliasing.
- What the compiler doesn’t know:
  - Which memory locations are “mutable shared” variables and could change asynchronously due to memory operations **in another thread**.
  - How to be conservative enough in the face of possible sharing.
- **Solution: Tell it.**
  - **Somehow identify the operations on “mutable shared” locations** (or equivalent information, but identifying shared variables is best).



Fact of Life

Software MMs have converged on
SC for data-race-free programs (SC-DRF).

Java: SC-DRF required since 2005.

C11 and C++11: SC-DRF default (relaxed == transitional tool).

Memory Model == Contract

You promise

To correctly **synchronize** your program (no race conditions).

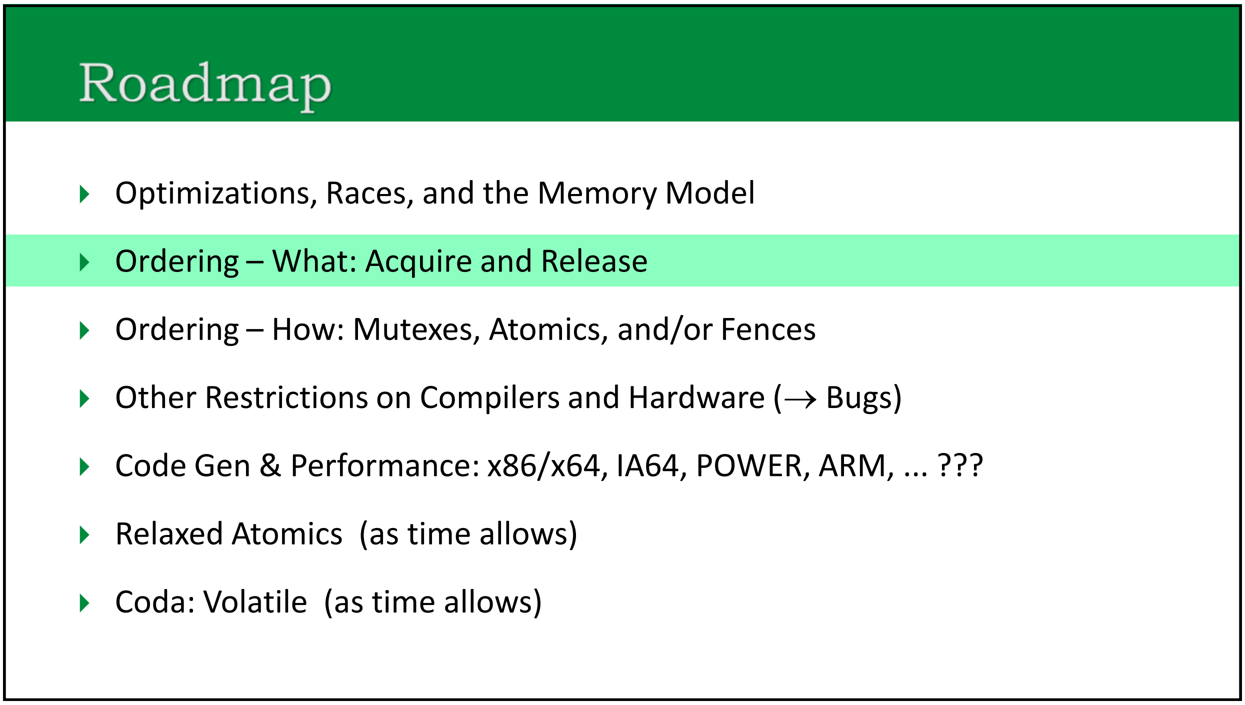
"The system" promises

To provide the illusion of executing the **program you wrote**.

How To Think About Races

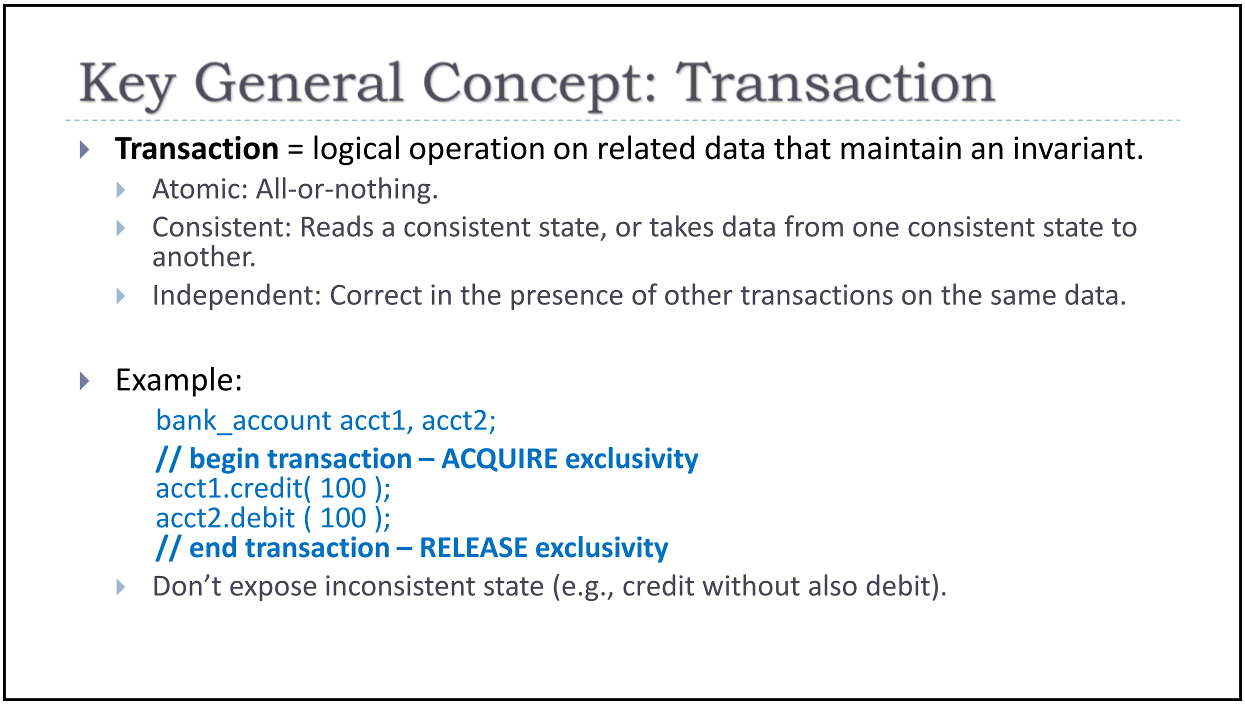
Q: While debugging an optimized build, have you ever seen pink elephants?

In a race, one thread can see into another thread **with the same view as a debugger**.



Roadmap

- Optimizations, Races, and the Memory Model
- Ordering – What: Acquire and Release
- Ordering – How: Mutexes, Atomics, and/or Fences
- Other Restrictions on Compilers and Hardware (→ Bugs)
- Code Gen & Performance: x86/x64, IA64, POWER, ARM, ... ???
- Relaxed Atomics (as time allows)
- Coda: Volatile (as time allows)

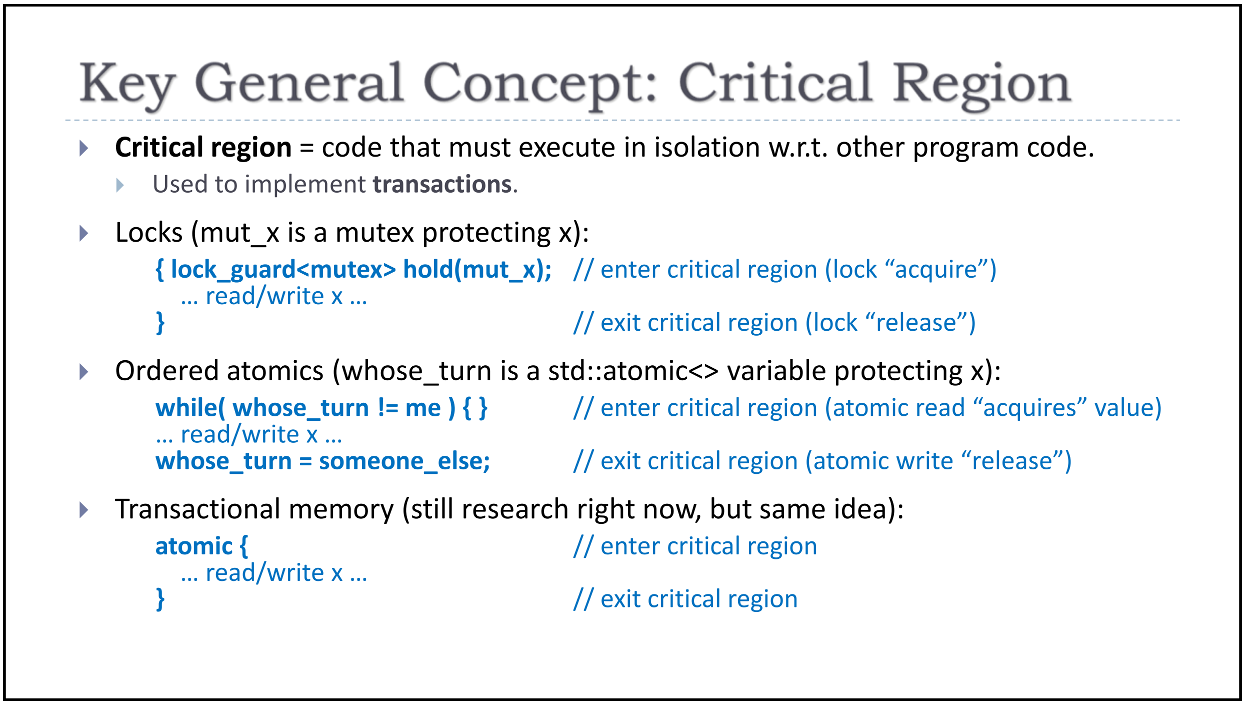


Key General Concept: Transaction

- **Transaction** = logical operation on related data that maintain an invariant.
  - Atomic: All-or-nothing.
  - Consistent: Reads a consistent state, or takes data from one consistent state to another.
  - Independent: Correct in the presence of other transactions on the same data.
- Example:

```
bank_account acct1, acct2;
// begin transaction – ACQUIRE exclusivity
acct1.credit( 100 );
acct2.debit ( 100 );
// end transaction – RELEASE exclusivity
```

  - Don’t expose inconsistent state (e.g., credit without also debit).



Key General Concept: Critical Region

- **Critical region** = code that must execute in isolation w.r.t. other program code.
  - Used to implement **transactions**.
- Locks (mut_x is a mutex protecting x):

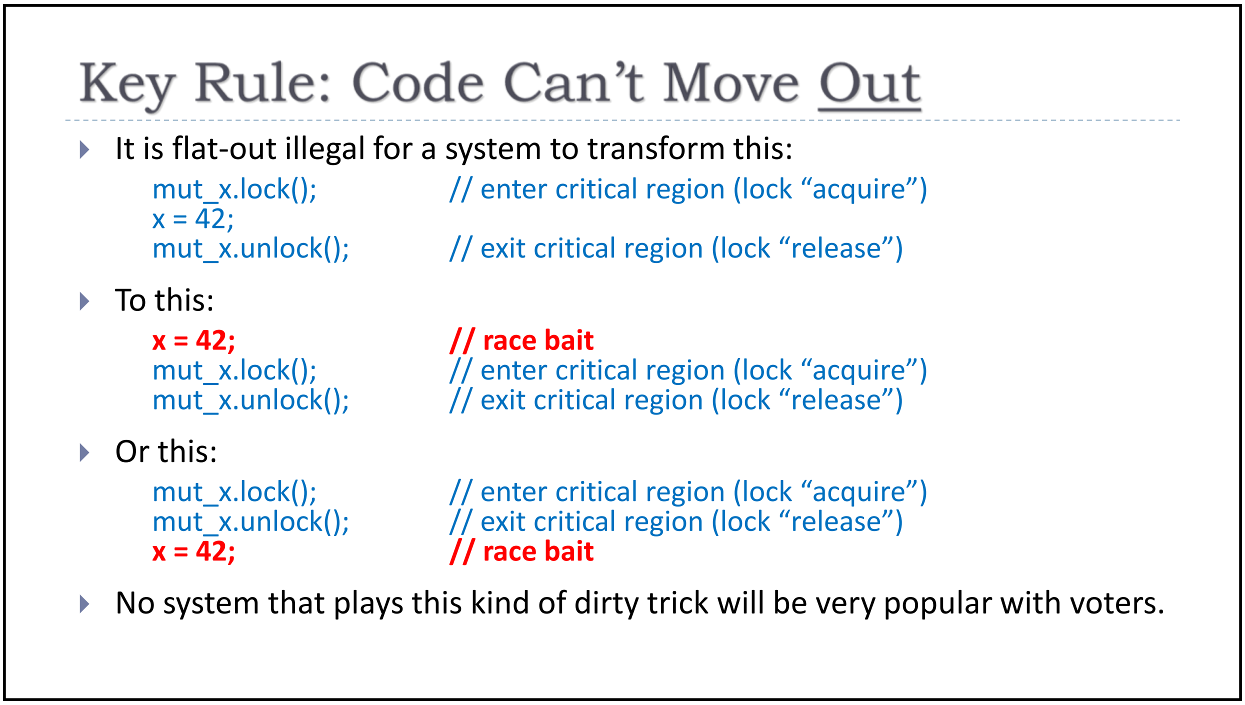
```
{ lock_guard<mutex> hold(mut_x);   // enter critical region (lock "acquire")
    … read/write x …
}                                  // exit critical region (lock "release")
```

- Ordered atomics (whose_turn is a std::atomic<> variable protecting x):

```
while( whose_turn != me ) { }      // enter critical region (atomic read "acquires" value)
… read/write x …
whose_turn = someone_else;         // exit critical region (atomic write "release")
```

- Transactional memory (still research right now, but same idea):

```
atomic {                           // enter critical region
    … read/write x …
}                                  // exit critical region
```



Key Rule: Code Can’t Move Out

- It is flat-out illegal for a system to transform this:

```
mut_x.lock();        // enter critical region (lock "acquire")
x = 42;
mut_x.unlock();      // exit critical region (lock "release")
```

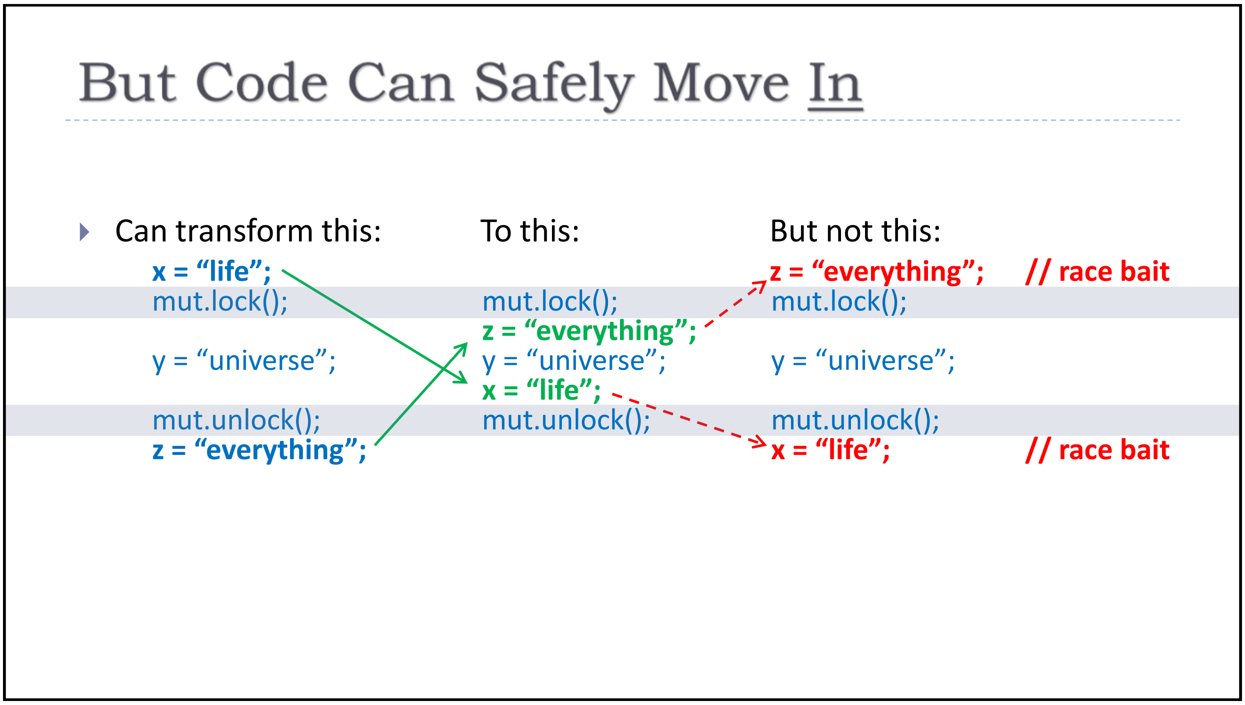
- To this:

```
x = 42;              // race bait
mut_x.lock();        // enter critical region (lock "acquire")
mut_x.unlock();      // exit critical region (lock "release")
```

- Or this:

```
mut_x.lock();        // enter critical region (lock "acquire")
mut_x.unlock();      // exit critical region (lock "release")
x = 42;              // race bait
```

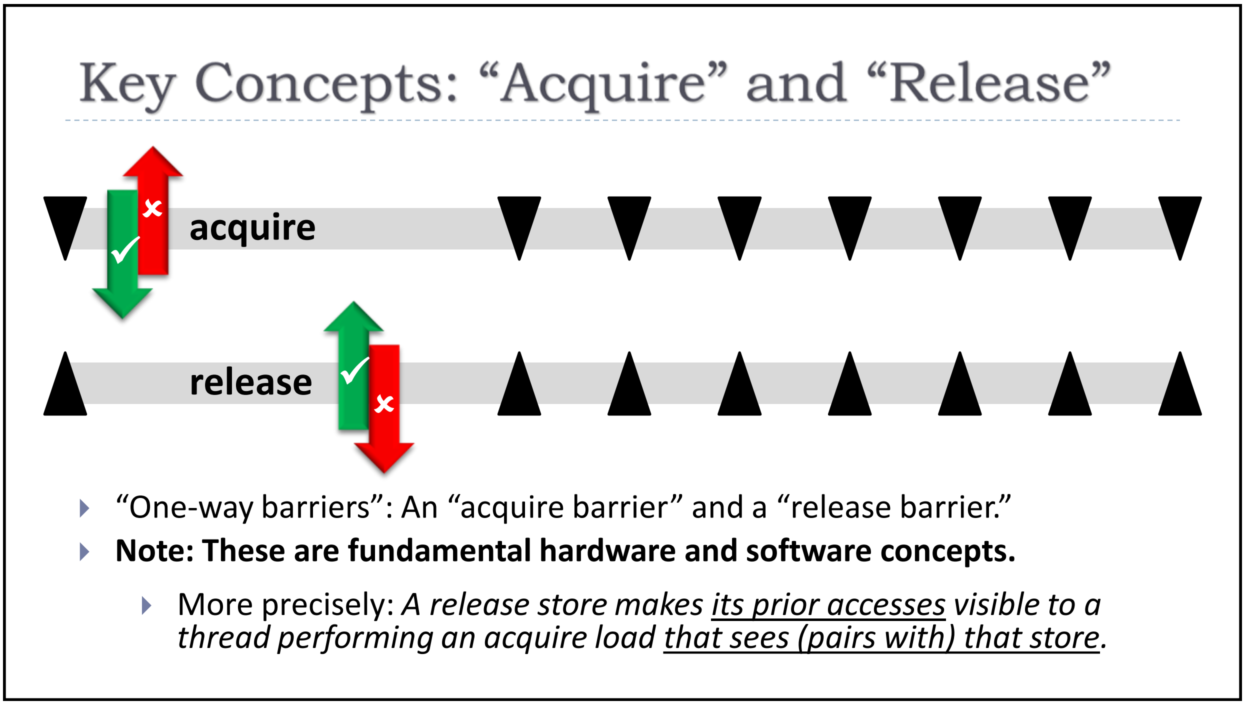
- No system that plays this kind of dirty trick will be very popular with voters.



But Code Can Safely Move In

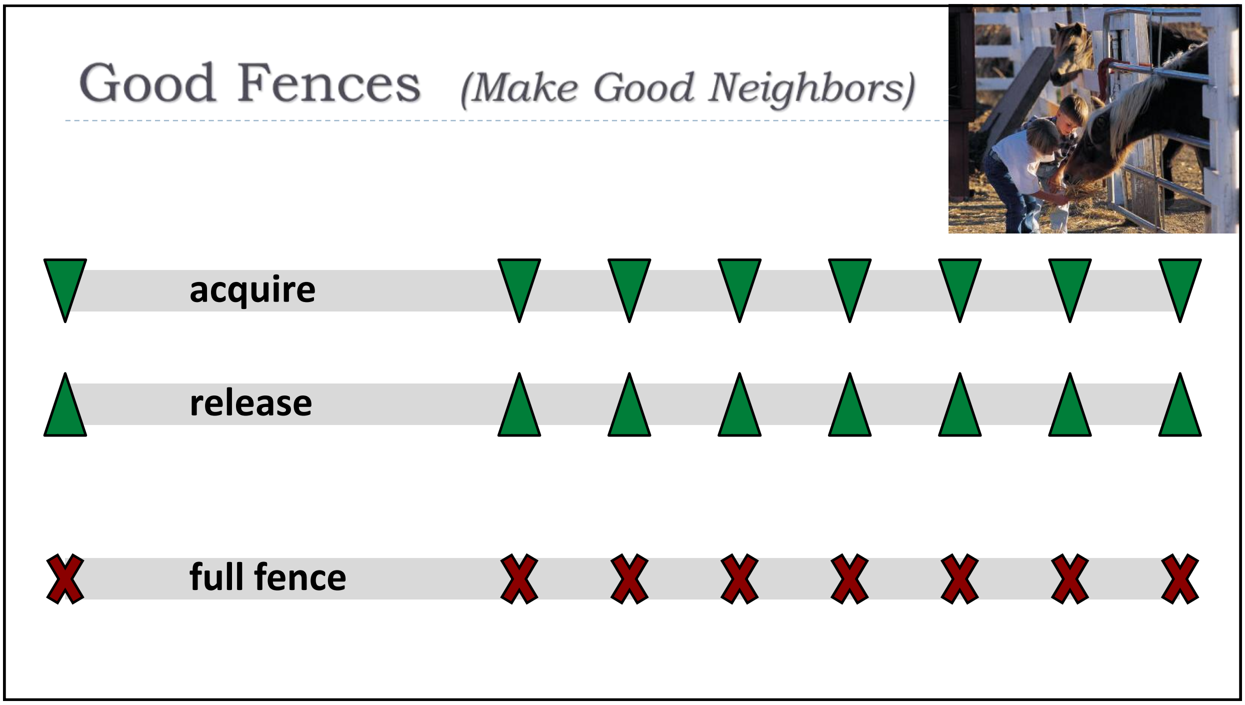
- Can transform this: To this: But not this:

```
x = "life";                                   z = "everything";    // race bait
mut.lock();            mut.lock();            mut.lock();
                       z = "everything";
y = "universe";        y = "universe";        y = "universe";
                       x = "life";
mut.unlock();          mut.unlock();          mut.unlock();
z = "everything";                             x = "life";          // race bait
```



Key Concepts: "Acquire" and "Release"

- "One-way barriers": An "acquire barrier" and a "release barrier."
- **Note: These are fundamental hardware and software concepts.**
  - More precisely: *A release store makes its prior accesses visible to a thread performing an acquire load that sees (pairs with) that store.*

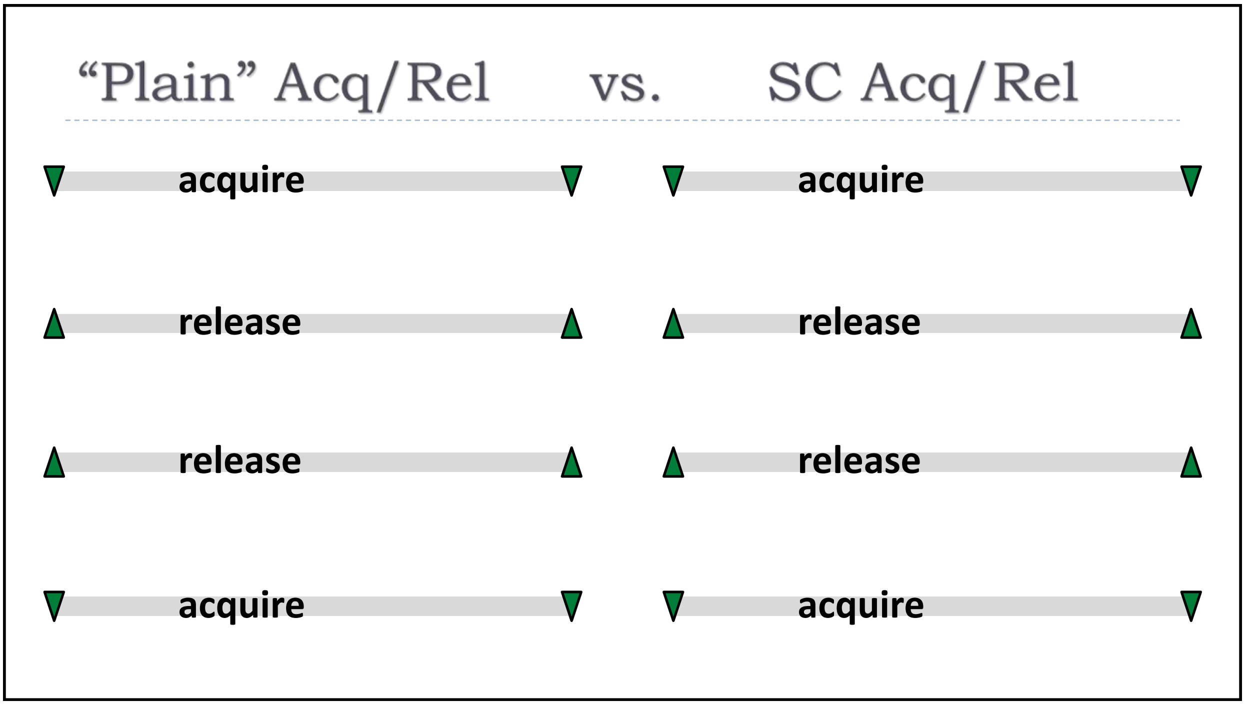


Good Fences *(Make Good Neighbors)*

▼ acquire ▼ ▼ ▼ ▼ ▼ ▼ ▼

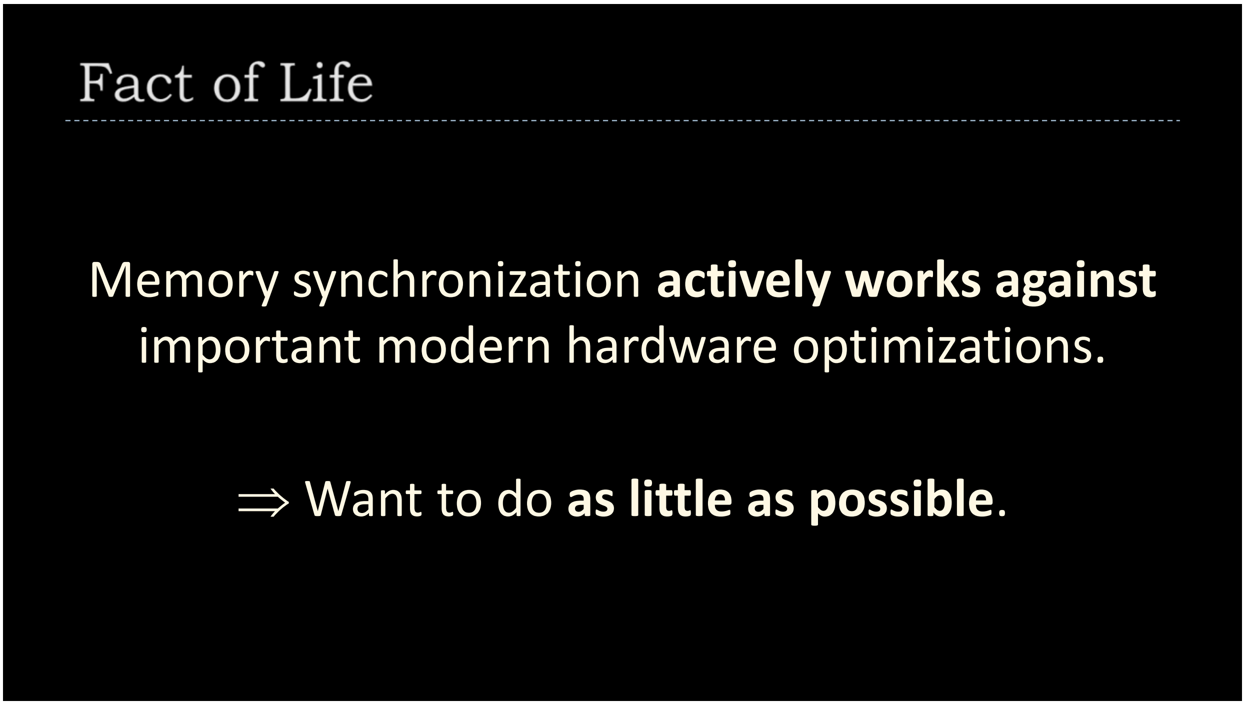
▲ release ▲ ▲ ▲ ▲ ▲ ▲ ▲

✗ full fence ✗ ✗ ✗ ✗ ✗ ✗ ✗



“Plain” Acq/Rel vs. SC Acq/Rel

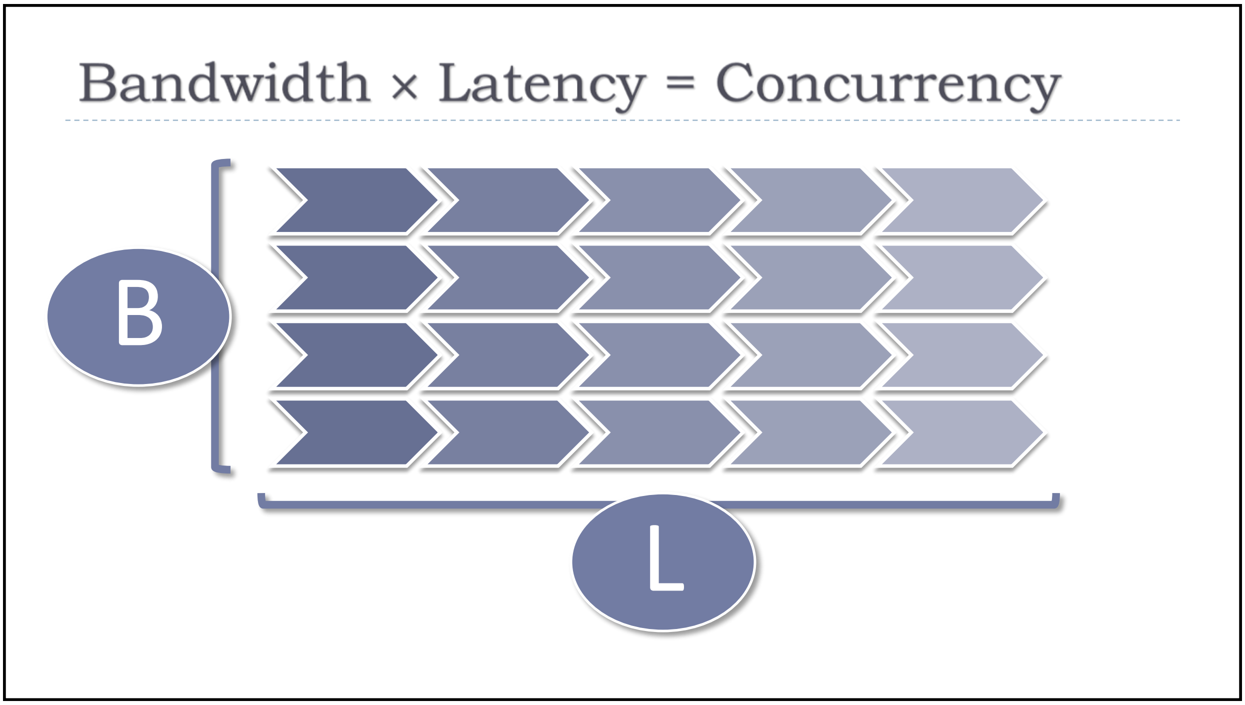
“Plain” Acq/Rel	SC Acq/Rel
▼ acquire ▼	▼ acquire ▼
▲ release ▲	▲ release ▲
▲ release ▲	▲ release ▲
▼ acquire ▼	▼ acquire ▼



Fact of Life

Memory synchronization **actively works against** important modern hardware optimizations.

⇒ Want to do **as little as possible**.

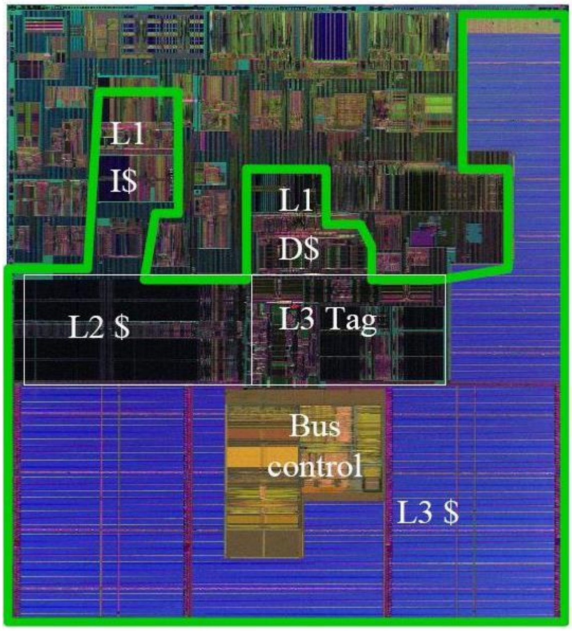


Bandwidth × Latency = Concurrency

Q: So How Do We Cope With Latency?
A: **Add Concurrency… Everywhere…**

Strategy	Technique	Can affect your code?
Parallelize (leverage compute power)	**Pipeline, execute out of order ("OoO"):** Launch expensive memory operations earlier, and do other work while waiting.	Yes
	Add <u>hardware</u> threads: Have other work available for the *same CPU core* to perform while other work is blocked on memory.	No *
Cache (leverage capacity)	**Instruction cache**	No
	Data cache: Multiple levels. Unit of sharing = cache line.	Yes
	Other buffering: Perhaps the most popular is store buffering, because writes are usually more expensive.	Yes
Speculate (leverage bandwidth, compute)	**Predict branches:** Guess whether an "if" will be true.	No
	Other optimistic execution: E.g., try both branches?	No
	Prefetch, scout: Warm up the cache.	No

** But you have to provide said other work (e.g., software threads) or this is useless!*



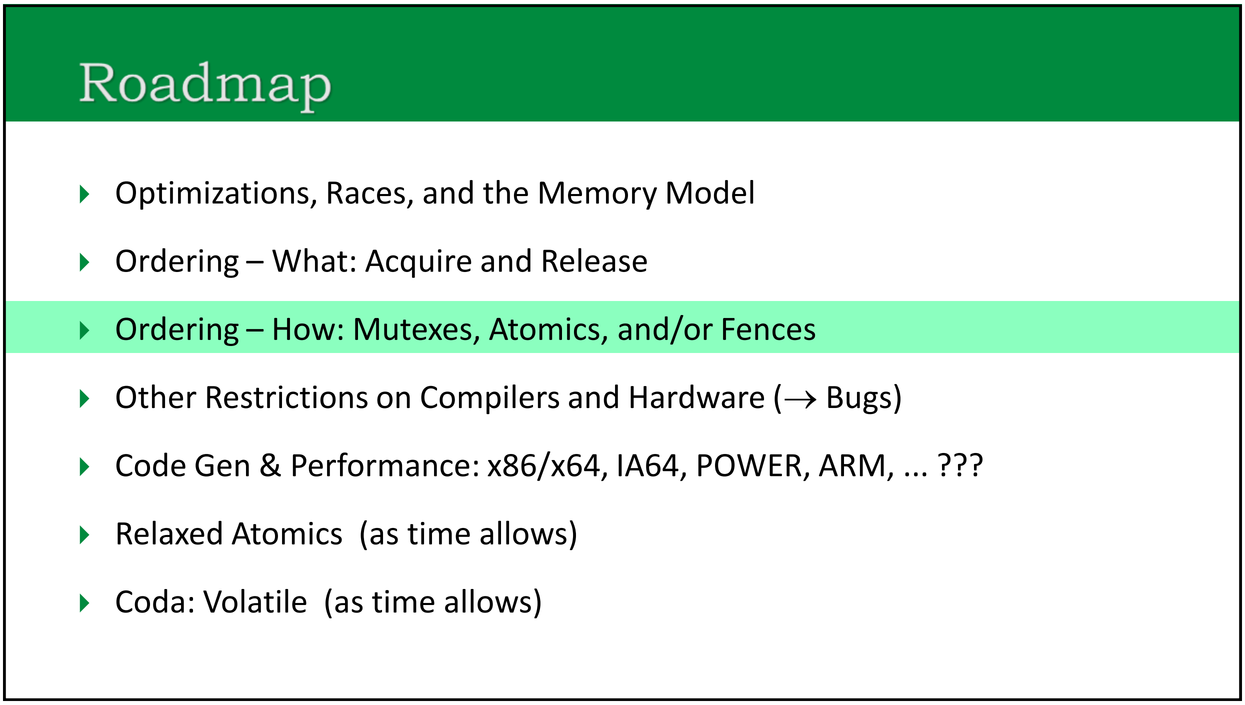
Sample Modern CPU

Original Itanium 2 had 211Mt, **85%** for cache:
16 KB L1I$ 16 KB L1D$
256 KB L2$ 3 MB L3$

1% of die to compute, **99%** to move/store data?

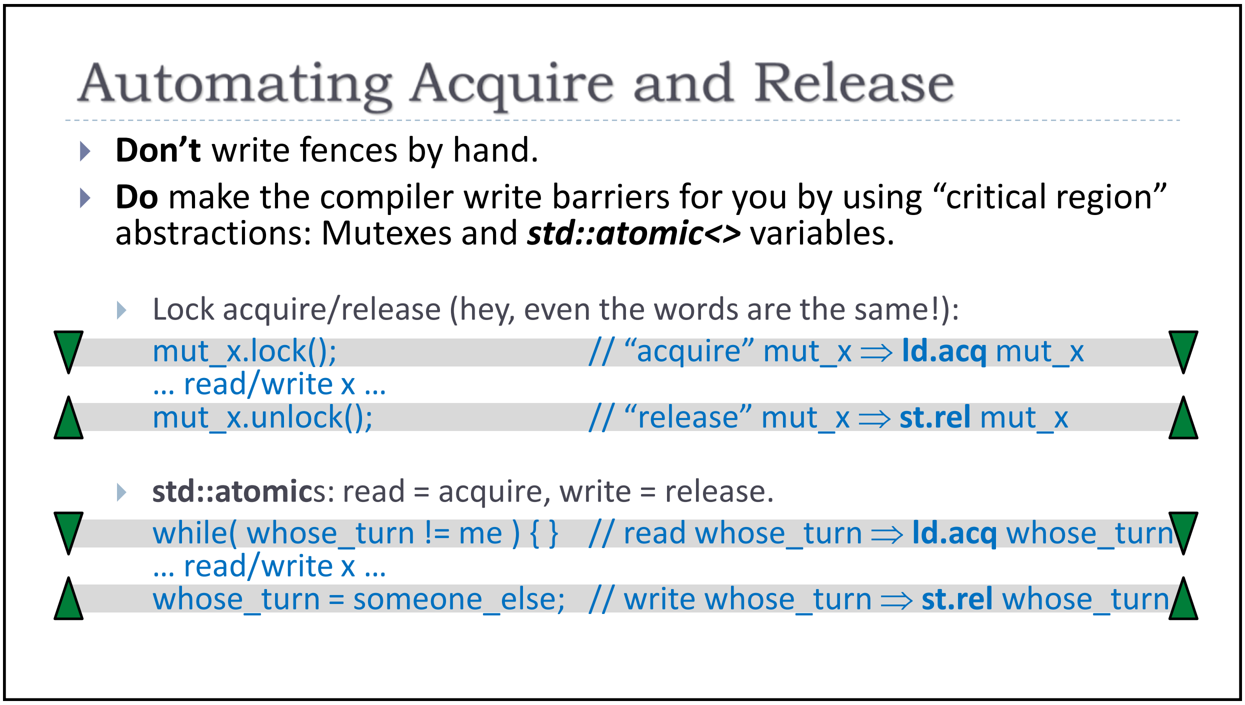
Itanium 2 9050:
Dual-core **24 MB L3$**

Source: David Patterson, UC Berkeley, HPEC keynote, Oct 2004
http://www.ll.mit.edu/HPEC/agendas/proc04/invited/patterson_keynote.pdf



Roadmap

- Optimizations, Races, and the Memory Model
- Ordering – What: Acquire and Release
- Ordering – How: Mutexes, Atomics, and/or Fences
- Other Restrictions on Compilers and Hardware (→ Bugs)
- Code Gen & Performance: x86/x64, IA64, POWER, ARM, ... ???
- Relaxed Atomics (as time allows)
- Coda: Volatile (as time allows)



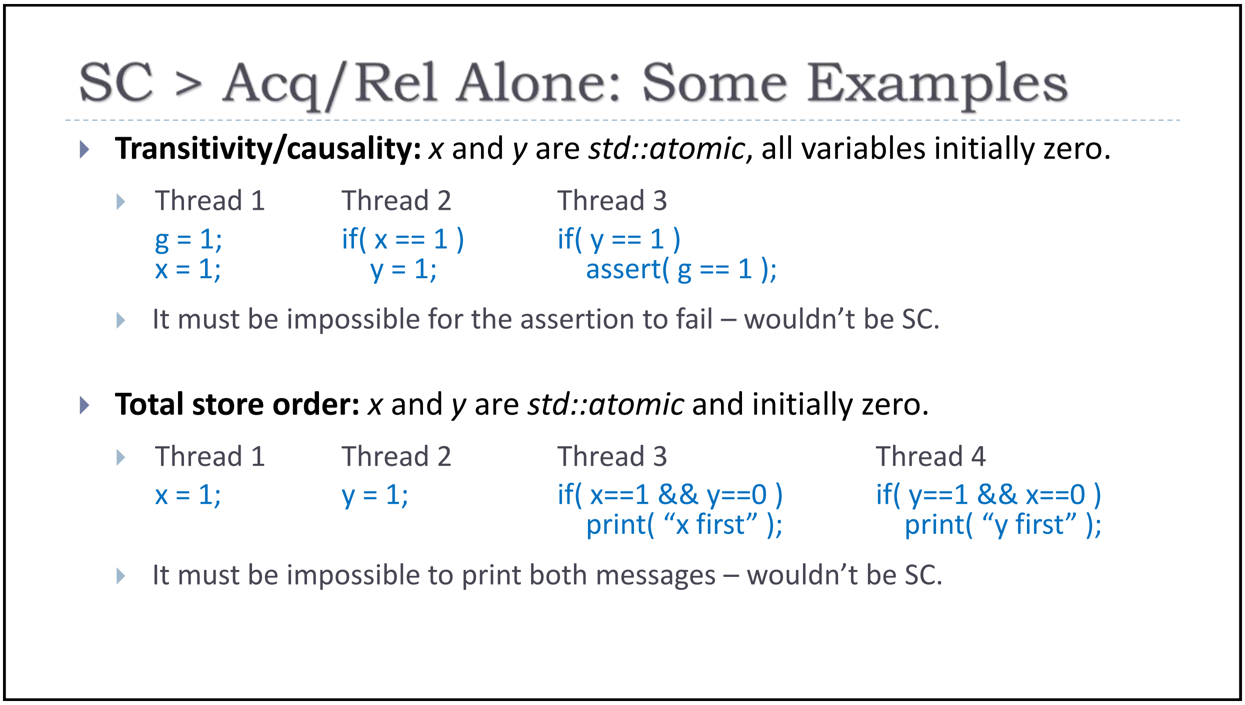
Automating Acquire and Release

- **Don't** write fences by hand.
- **Do** make the compiler write barriers for you by using "critical region" abstractions: Mutexes and ***std::atomic<>*** variables.
  - Lock acquire/release (hey, even the words are the same!):

```
▼ mut_x.lock();                      // "acquire" mut_x ⇒ ld.acq mut_x   ▼
  … read/write x …
▲ mut_x.unlock();                    // "release" mut_x ⇒ st.rel mut_x   ▲
```

  - **std::atomic**s: read = acquire, write = release.

```
▼ while( whose_turn != me ) { }   // read whose_turn ⇒ ld.acq whose_turn  ▼
  … read/write x …
▲ whose_turn = someone_else;      // write whose_turn ⇒ st.rel whose_turn ▲
```



SC > Acq/Rel Alone: Some Examples

- **Transitivity/causality:** *x* and *y* are *std::atomic*, all variables initially zero.

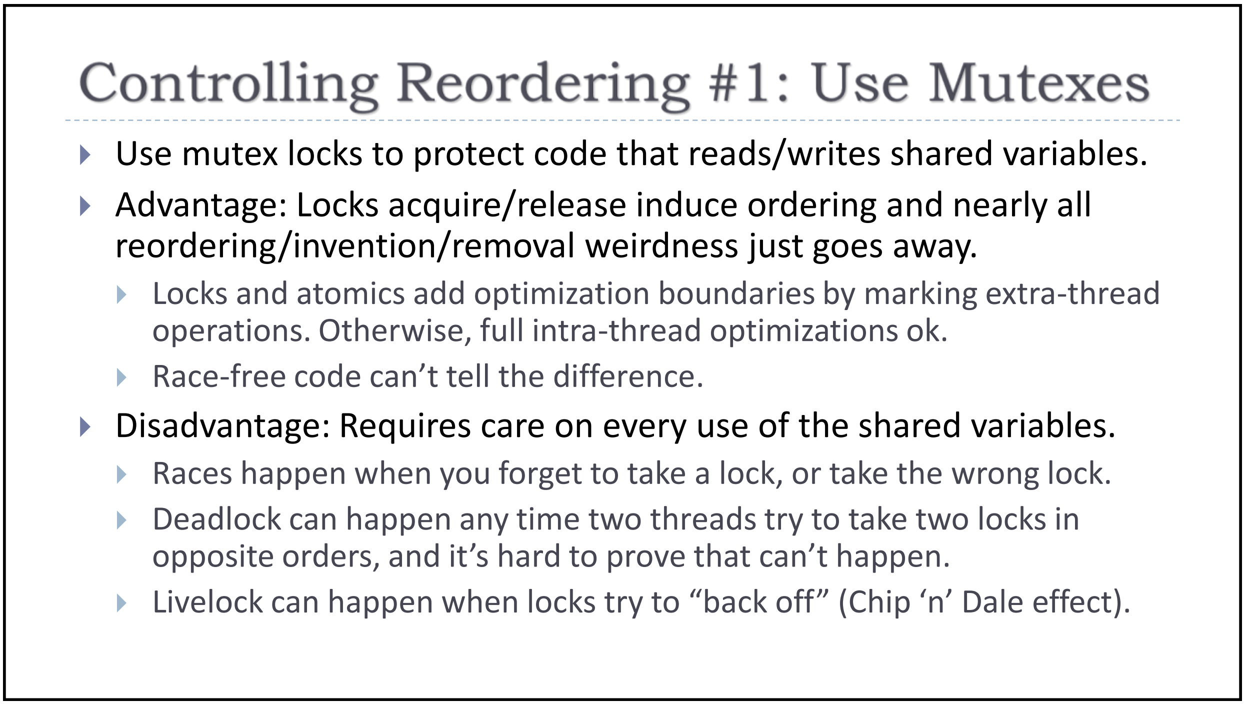
```
Thread 1      Thread 2        Thread 3
g = 1;        if( x == 1 )    if( y == 1 )
x = 1;          y = 1;          assert( g == 1 );
```

  - It must be impossible for the assertion to fail – wouldn't be SC.

- **Total store order:** *x* and *y* are *std::atomic* and initially zero.

```
Thread 1      Thread 2        Thread 3                  Thread 4
x = 1;        y = 1;          if( x==1 && y==0 )        if( y==1 && x==0 )
                                print( "x first" );       print( "y first" );
```

  - It must be impossible to print both messages – wouldn't be SC.



Controlling Reordering #1: Use Mutexes

- Use mutex locks to protect code that reads/writes shared variables.
- Advantage: Locks acquire/release induce ordering and nearly all reordering/invention/removal weirdness just goes away.
  - Locks and atomics add optimization boundaries by marking extra-thread operations. Otherwise, full intra-thread optimizations ok.
  - Race-free code can't tell the difference.
- Disadvantage: Requires care on every use of the shared variables.
  - Races happen when you forget to take a lock, or take the wrong lock.
  - Deadlock can happen any time two threads try to take two locks in opposite orders, and it's hard to prove that can't happen.
  - Livelock can happen when locks try to "back off" (Chip 'n' Dale effect).

Controlling Reordering #2: std::atomic<>

- *Special atomic types* are automatically safe from reordering:

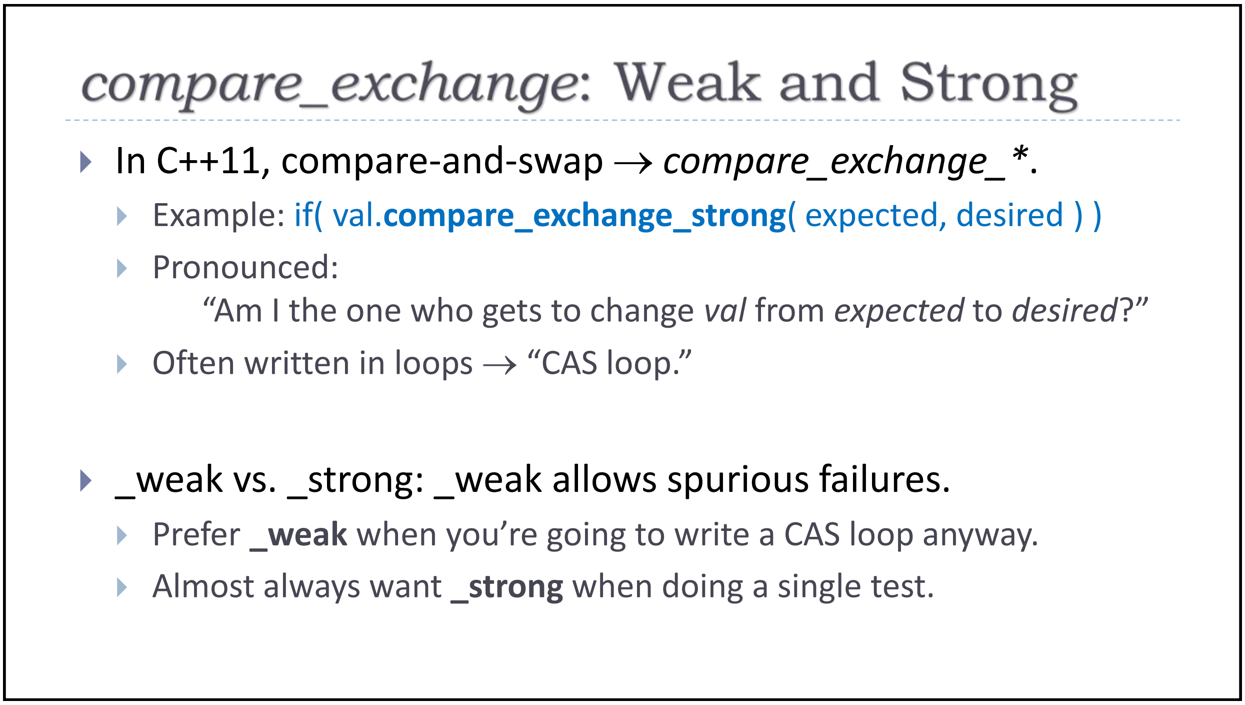
```
atomic<int> flag1 = 0, flag2 = 0;
// From Dekker's algorithm
flag1 = 1;
if( flag2 != 0 ) { ... }
```

- Advantage: Just tag the variable, not every place it's used.
- Disadvantage: Writing correct atomics code is harder than it looks.
  - A new 'lock-free' algorithm or data structure is a publishable result.
  - Some common data structures have no known *practical* 'lock-free' implementation.

Ordered Atomics

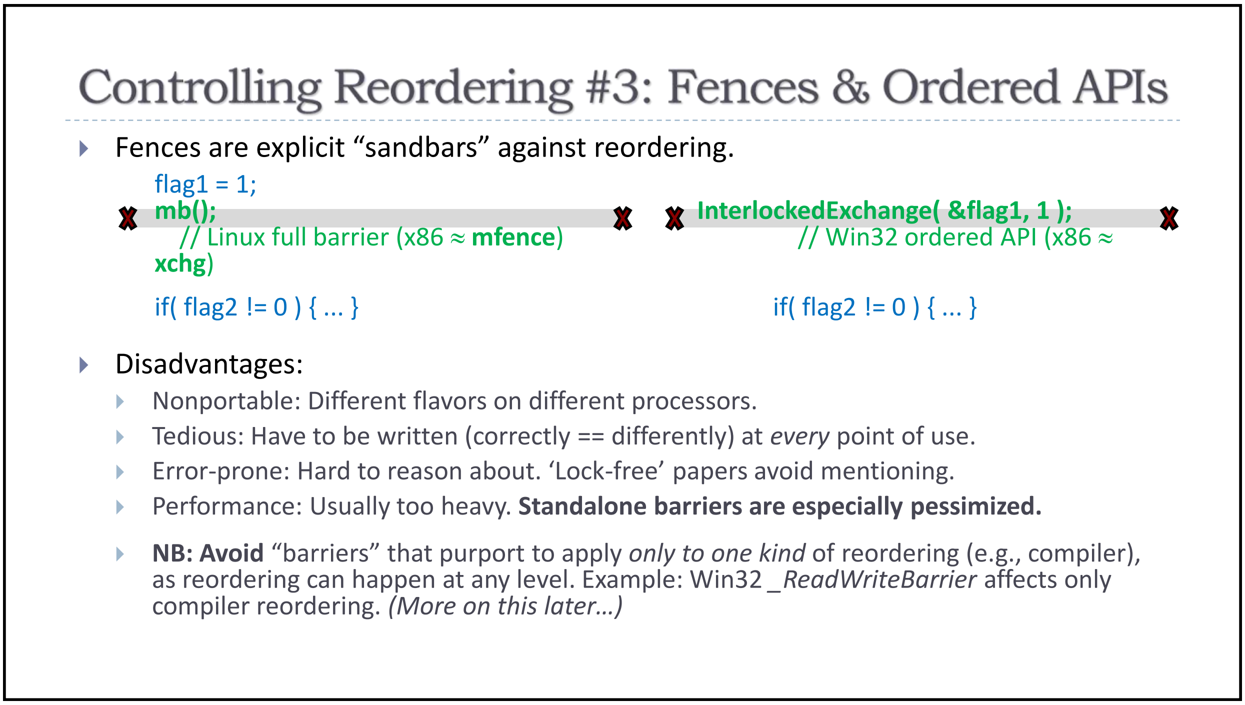
- Ordered atomic variables:
  - Java and ≈.NET *volatile*. — **Always SC.** *(NB: Java .lazySet not precisely defined.)*
  - C++11 *atomic<T>*, C11 atomic_*. — **Default SC.** *(NB: Not C/C++ volatile!)*
- Semantics and operations:
  - Each individual read/write is **atomic**. No torn reads, no locking needed.
  - Each thread's reads/writes are guaranteed to execute in **order**.
  - **Special ops: [Compare-and-]swap (CAS).** Conceptually atomic execution of:

```
T atomic<T>::exchange( T desired )
   {  T oldval = this->value;  this->value = desired;  return oldval; }

bool atomic<T>::compare_exchange_strong( T& expected, T desired ) {
   if( this->value == expected ) { this->value = desired; return true; }
   expected = this->value; return false;
}
```



compare_exchange: Weak and Strong

- In C++11, compare-and-swap → *compare_exchange_*.*
  - Example: if(val.**compare_exchange_strong**(expected, desired))
  - Pronounced:
    "Am I the one who gets to change *val* from *expected* to *desired*?"
  - Often written in loops → "CAS loop."
- _weak vs. _strong: _weak allows spurious failures.
  - Prefer **_weak** when you're going to write a CAS loop anyway.
  - Almost always want **_strong** when doing a single test.

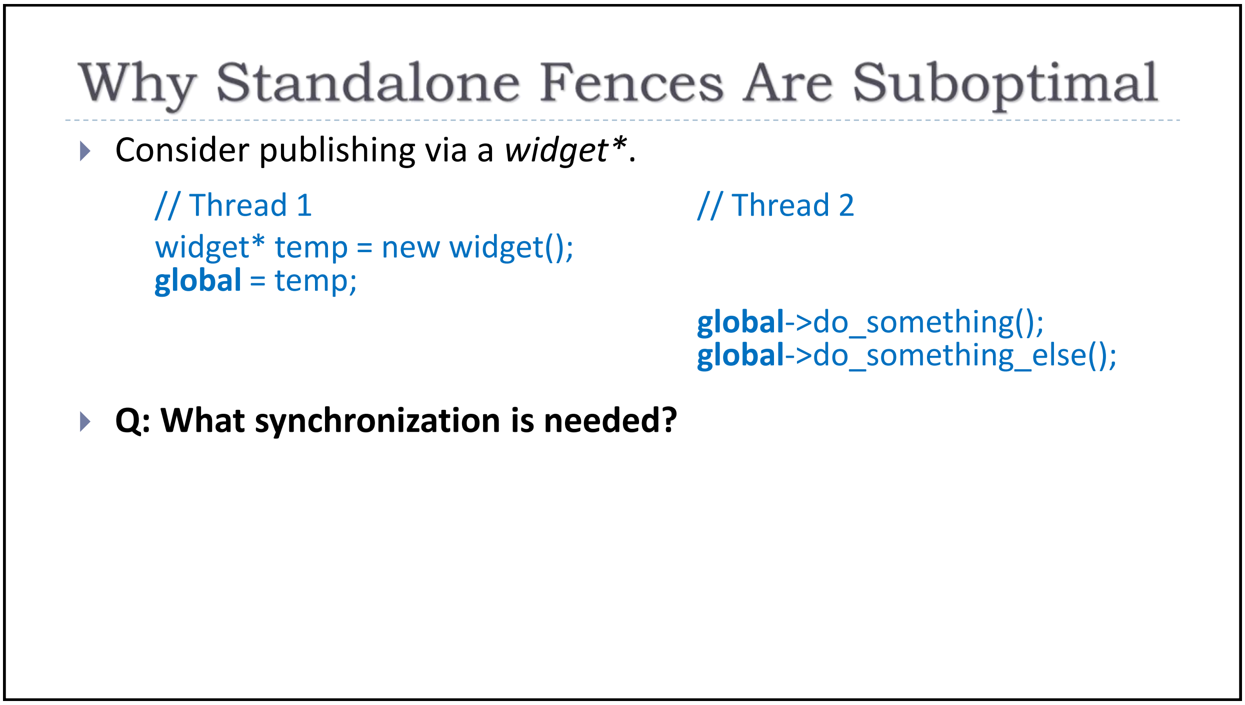


Controlling Reordering #3: Fences & Ordered APIs

- Fences are explicit "sandbars" against reordering.

```
flag1 = 1;
✗ mb(); ✗                                   ✗ InterlockedExchange( &flag1, 1 ); ✗
   // Linux full barrier (x86 ≈ mfence)           // Win32 ordered API (x86 ≈
xchg)

if( flag2 != 0 ) { ... }                    if( flag2 != 0 ) { ... }
```

- Disadvantages:
  - Nonportable: Different flavors on different processors.
  - Tedious: Have to be written (correctly == differently) at *every* point of use.
  - Error-prone: Hard to reason about. 'Lock-free' papers avoid mentioning.
  - Performance: Usually too heavy. **Standalone barriers are especially pessimized.**
  - **NB: Avoid** "barriers" that purport to apply *only to one kind* of reordering (e.g., compiler), as reordering can happen at any level. Example: Win32 _*ReadWriteBarrier* affects only compiler reordering. *(More on this later…)*

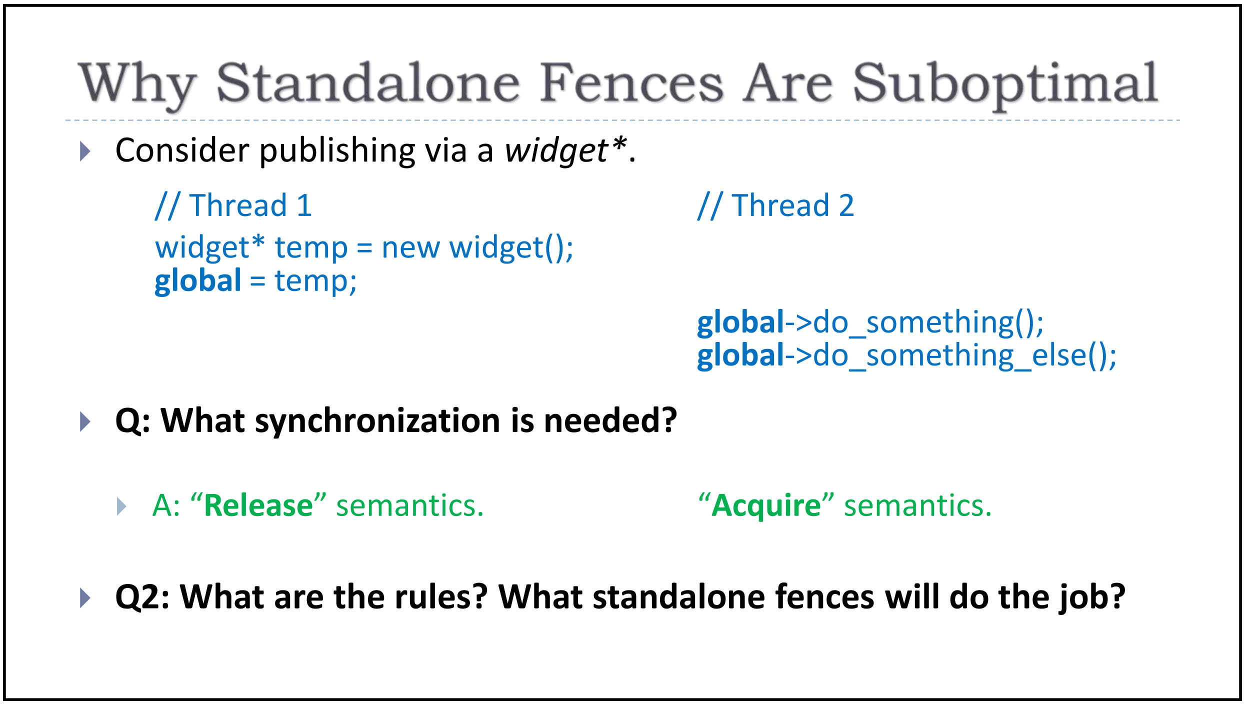


Why Standalone Fences Are Suboptimal

- Consider publishing via a *widget**.

```
// Thread 1                          // Thread 2
widget* temp = new widget();
global = temp;
                                     global->do_something();
                                     global->do_something_else();
```

- **Q: What synchronization is needed?**

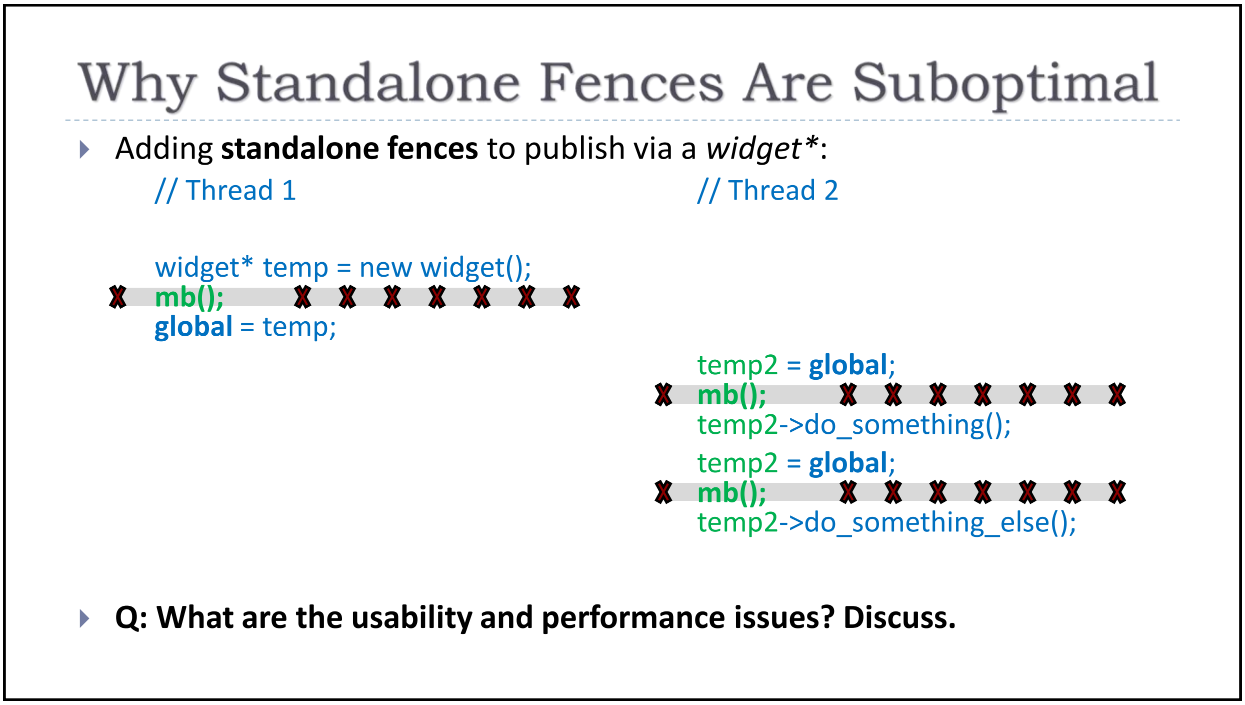


Why Standalone Fences Are Suboptimal

- Consider publishing via a *widget**.

```
// Thread 1                          // Thread 2
widget* temp = new widget();
global = temp;
                                     global->do_something();
                                     global->do_something_else();
```

- **Q: What synchronization is needed?**
  - A: "**Release**" semantics. "**Acquire**" semantics.
- **Q2: What are the rules? What standalone fences will do the job?**

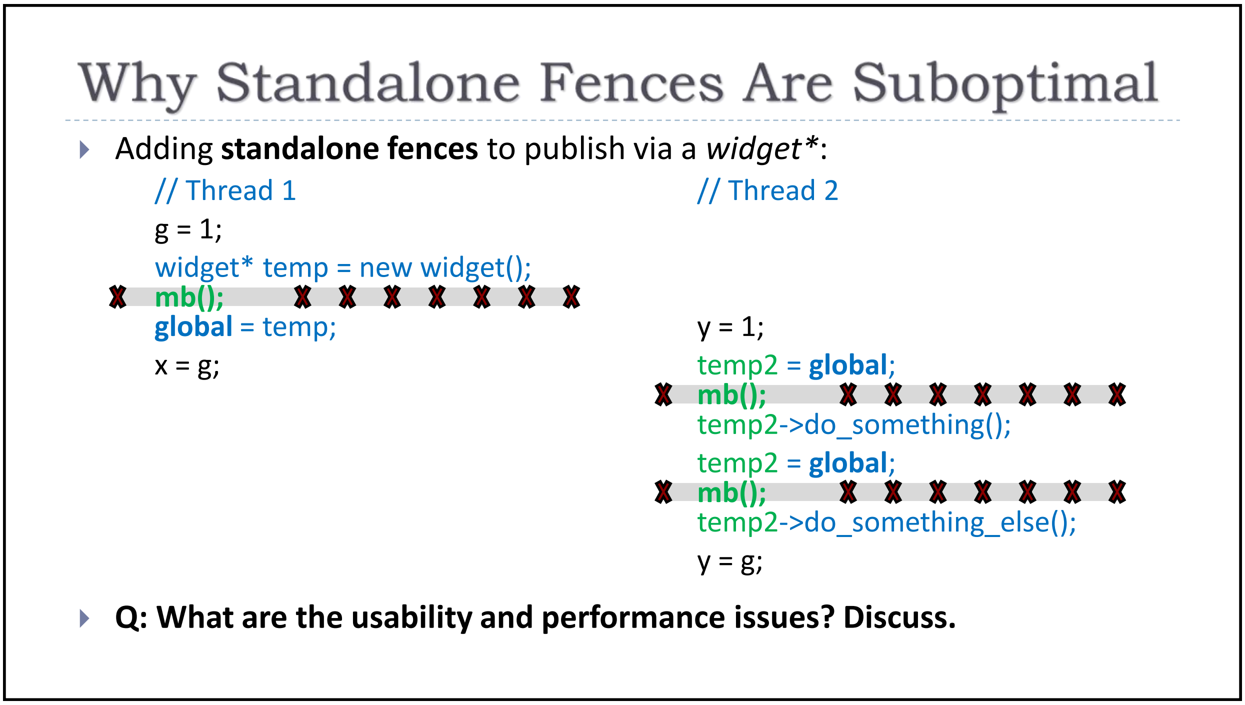


Why Standalone Fences Are Suboptimal

- Adding **standalone fences** to publish via a *widget**:

```
    // Thread 1                              // Thread 2


    widget* temp = new widget();
  ✗ mb();      ✗  ✗  ✗  ✗  ✗  ✗  ✗
    global = temp;

                                             temp2 = global;
                                           ✗ mb();      ✗  ✗  ✗  ✗  ✗  ✗  ✗
                                             temp2->do_something();
                                             temp2 = global;
                                           ✗ mb();      ✗  ✗  ✗  ✗  ✗  ✗  ✗
                                             temp2->do_something_else();
```

- **Q: What are the usability and performance issues? Discuss.**

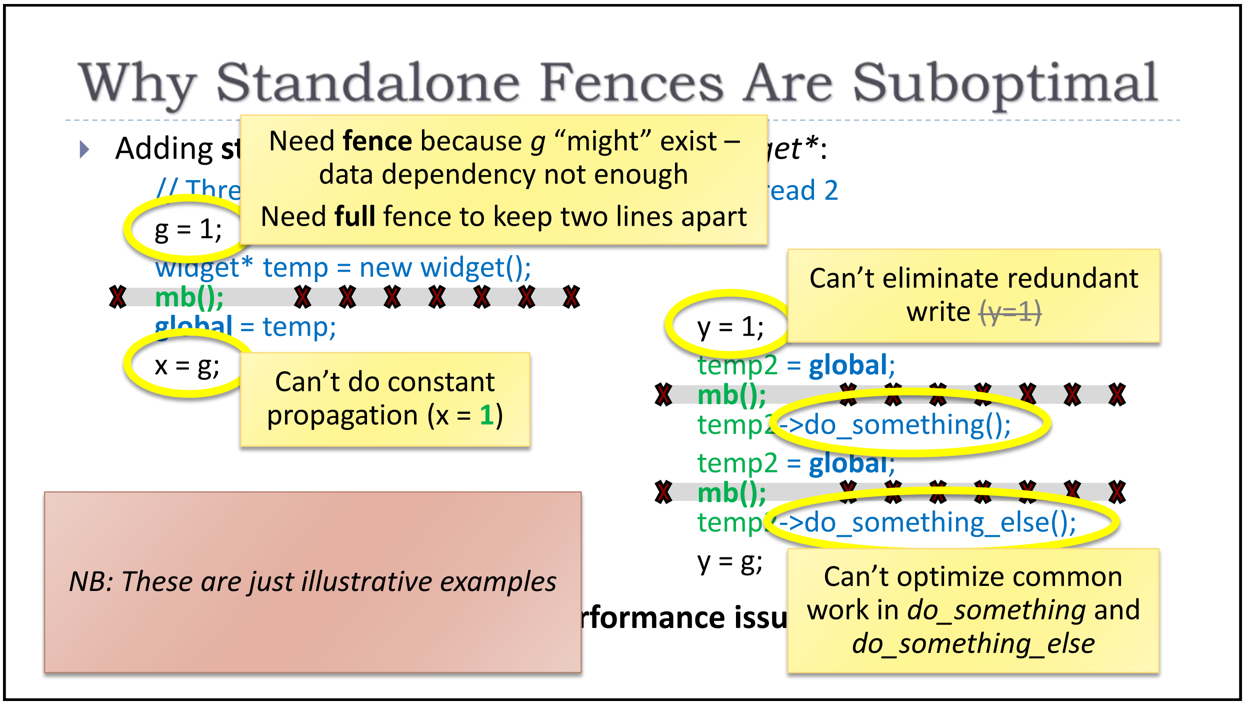


Why Standalone Fences Are Suboptimal

- Adding **standalone fences** to publish via a *widget**:

```
    // Thread 1                              // Thread 2
    g = 1;
    widget* temp = new widget();
  ✗ mb();      ✗  ✗  ✗  ✗  ✗  ✗  ✗
    global = temp;                           y = 1;
    x = g;                                   temp2 = global;
                                           ✗ mb();      ✗  ✗  ✗  ✗  ✗  ✗  ✗
                                             temp2->do_something();
                                             temp2 = global;
                                           ✗ mb();      ✗  ✗  ✗  ✗  ✗  ✗  ✗
                                             temp2->do_something_else();
                                             y = g;
```

- **Q: What are the usability and performance issues? Discuss.**



Why Standalone Fences Are Suboptimal

- Adding st ... et*:

```
// Thre                                     ead 2
g = 1;
widget* temp = new widget();
mb();
global = temp;
x = g;
                                 y = 1;
                                 temp2 = global;
                                 mb();
                                 temp2->do_something();
                                 temp2 = global;
                                 mb();
                                 temp2->do_something_else();
                                 y = g;
```

Need **fence** because *g* “might” exist – data dependency not enough

Need **full** fence to keep two lines apart

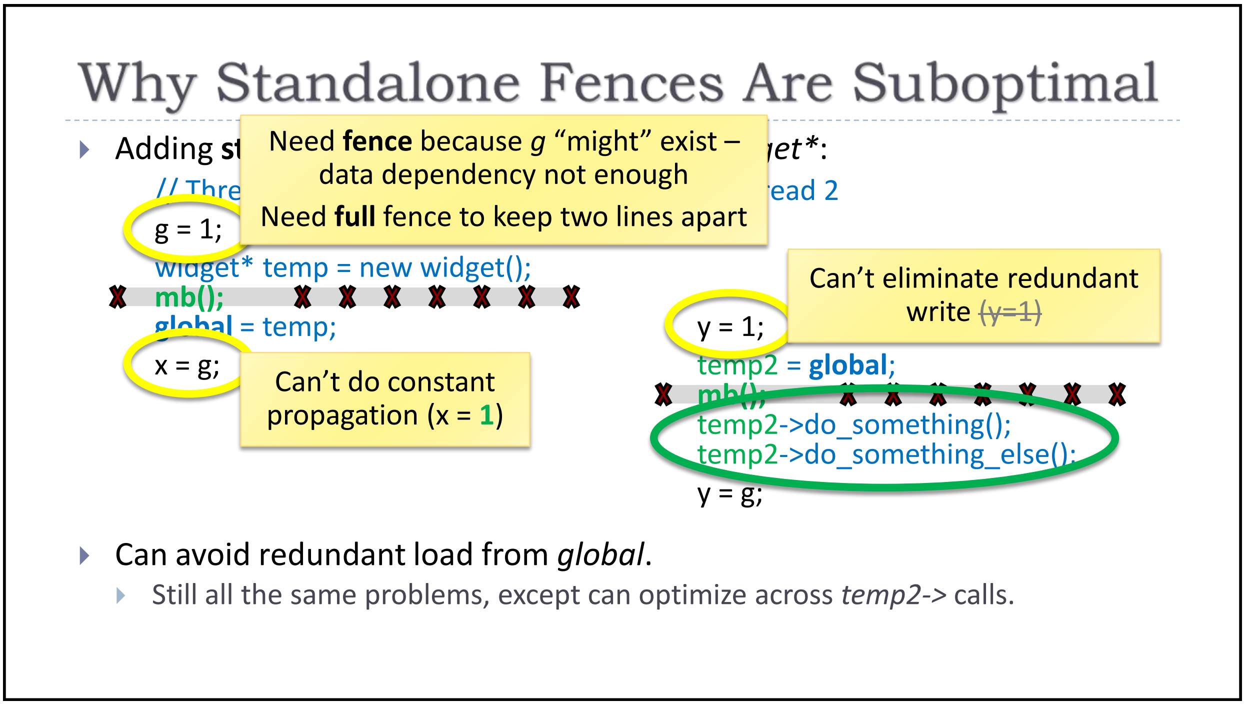
Can’t eliminate redundant write ~~(y=1)~~

Can’t do constant propagation (x = 1)

Can’t optimize common work in *do_something* and *do_something_else*

NB: These are just illustrative examples

rformance issu



Why Standalone Fences Are Suboptimal

- Adding st ... et*:

```
// Thre                                     ead 2
g = 1;
widget* temp = new widget();
mb();
global = temp;
x = g;
                                 y = 1;
                                 temp2 = global;
                                 mb();
                                 temp2->do_something();
                                 temp2->do_something_else();
                                 y = g;
```

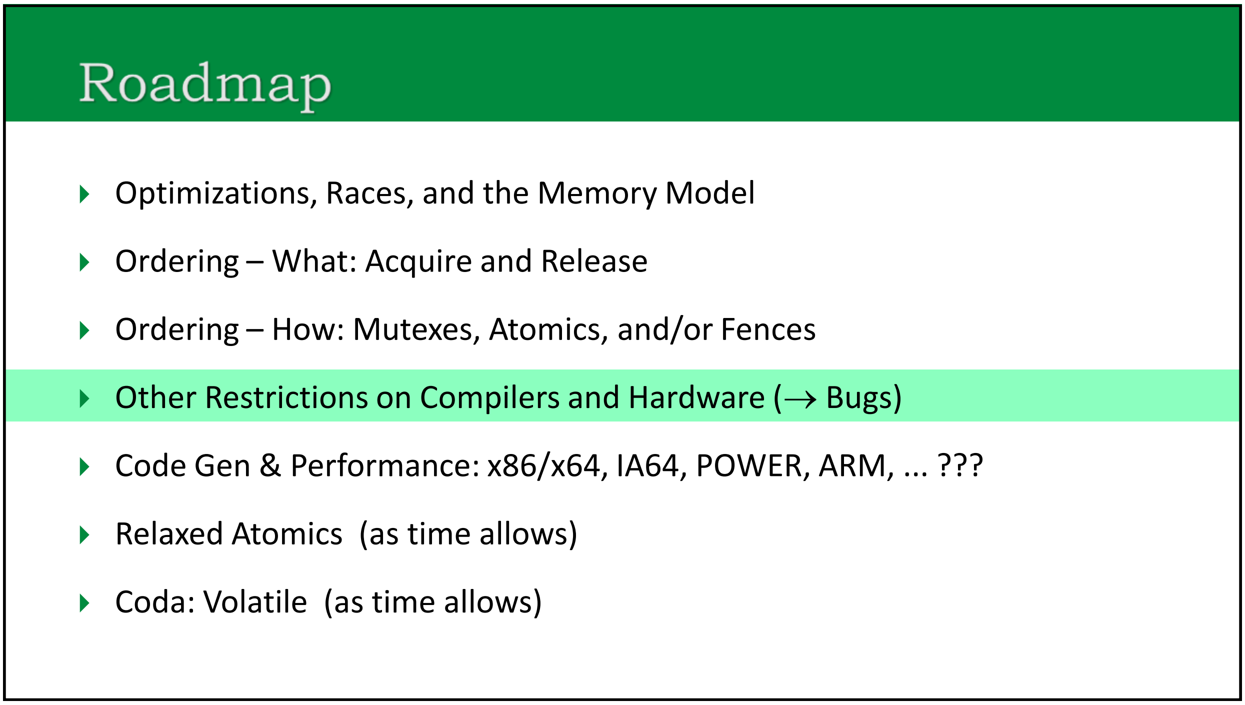
Need **fence** because *g* “might” exist – data dependency not enough

Need **full** fence to keep two lines apart

Can’t eliminate redundant write ~~(y=1)~~

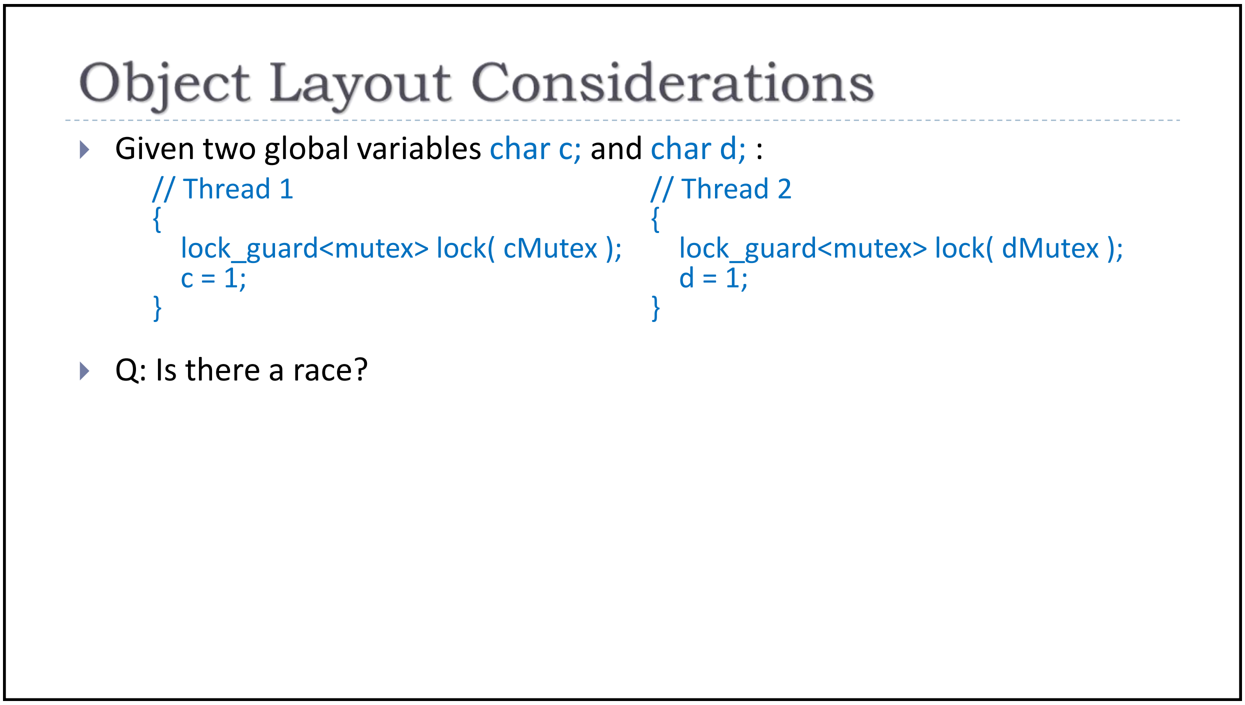
Can’t do constant propagation (x = 1)

- Can avoid redundant load from *global*.
  - Still all the same problems, except can optimize across *temp2->* calls.



Roadmap

- Optimizations, Races, and the Memory Model
- Ordering – What: Acquire and Release
- Ordering – How: Mutexes, Atomics, and/or Fences
- Other Restrictions on Compilers and Hardware (→ Bugs)
- Code Gen & Performance: x86/x64, IA64, POWER, ARM, ... ???
- Relaxed Atomics (as time allows)
- Coda: Volatile (as time allows)

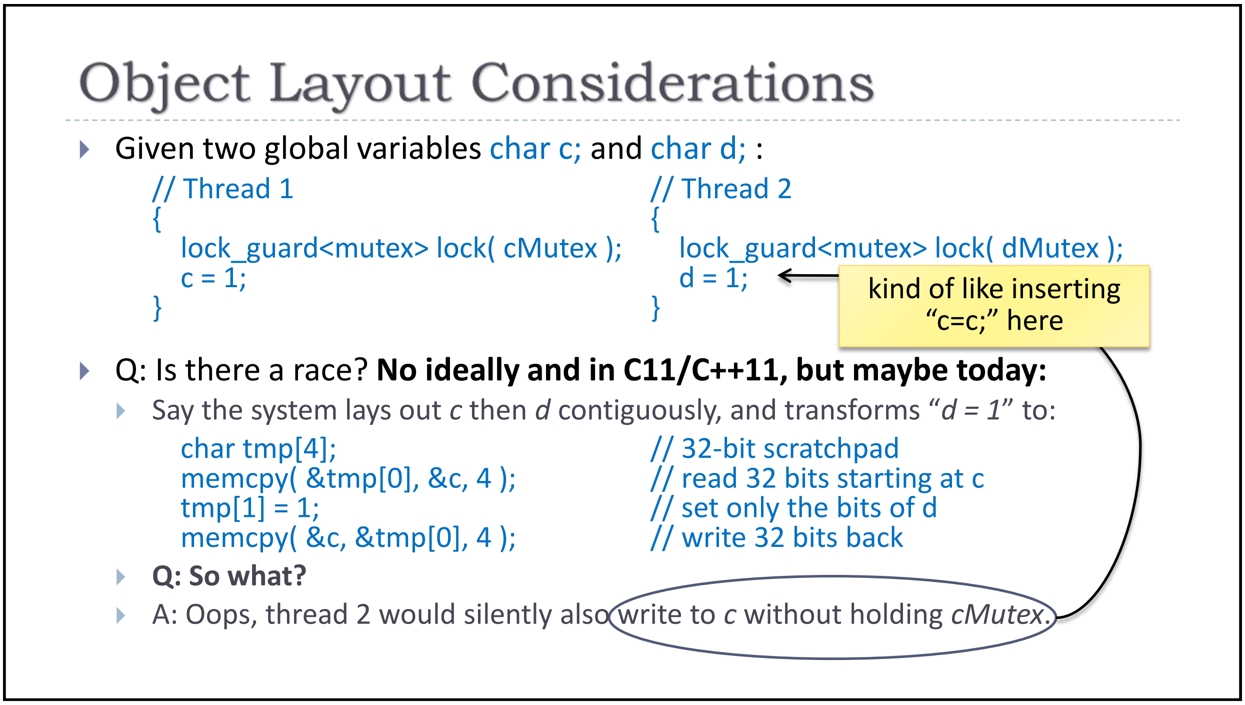


Object Layout Considerations

- Given two global variables char c; and char d; :

```
// Thread 1                              // Thread 2
{                                        {
   lock_guard<mutex> lock( cMutex );        lock_guard<mutex> lock( dMutex );
   c = 1;                                   d = 1;
}                                        }
```

- Q: Is there a race?



Object Layout Considerations

- Given two global variables char c; and char d; :

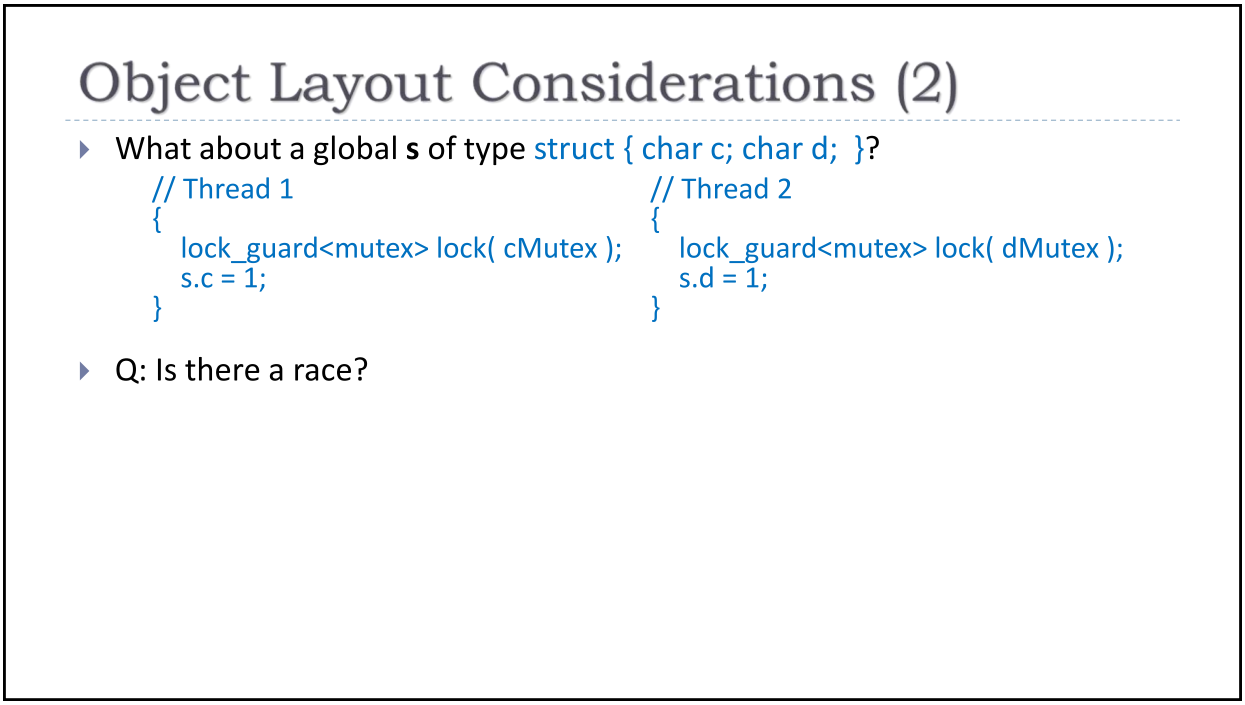
```
// Thread 1                          // Thread 2
{                                    {
   lock_guard<mutex> lock( cMutex );    lock_guard<mutex> lock( dMutex );
   c = 1;                               d = 1;
}                                    }
```

kind of like inserting "c=c;" here

- Q: Is there a race? **No ideally and in C11/C++11, but maybe today:**
  - Say the system lays out *c* then *d* contiguously, and transforms "*d = 1*" to:

```
char tmp[4];                         // 32-bit scratchpad
memcpy( &tmp[0], &c, 4 );            // read 32 bits starting at c
tmp[1] = 1;                          // set only the bits of d
memcpy( &c, &tmp[0], 4 );            // write 32 bits back
```

  - **Q: So what?**
  - A: Oops, thread 2 would silently also write to *c* without holding *cMutex*.

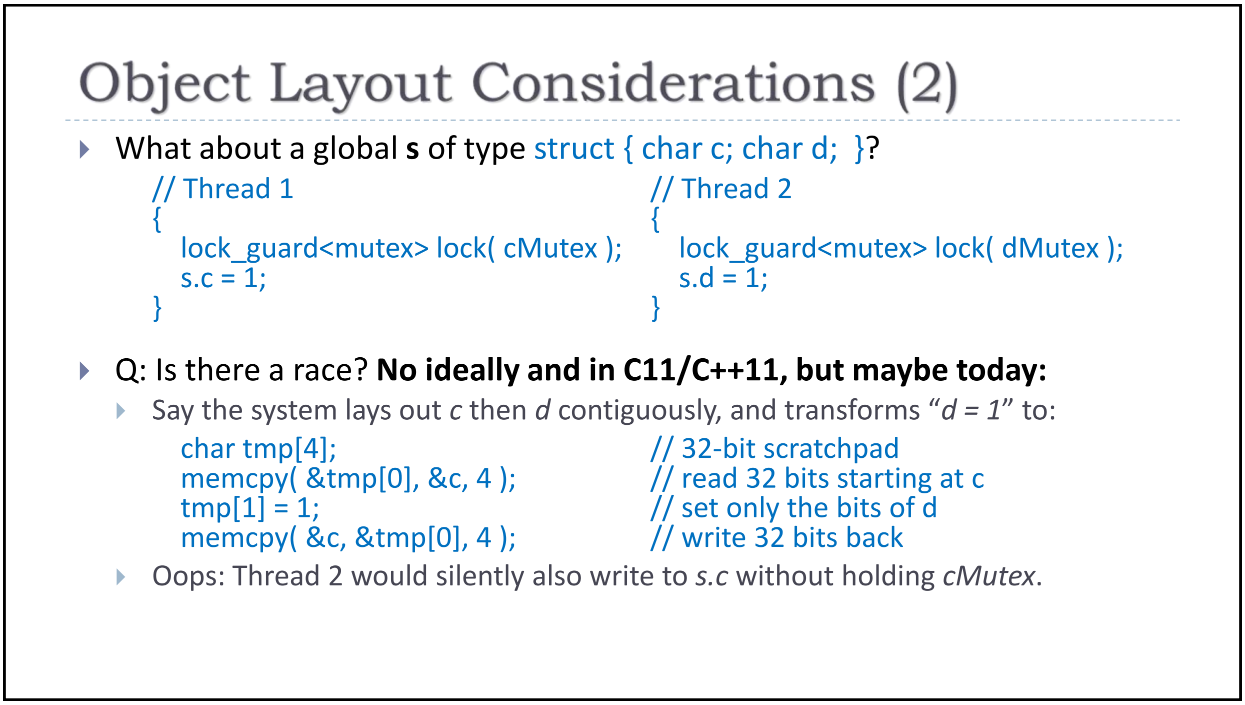


Object Layout Considerations (2)

- What about a global **s** of type struct { char c; char d; }?

```
// Thread 1                          // Thread 2
{                                    {
   lock_guard<mutex> lock( cMutex );    lock_guard<mutex> lock( dMutex );
   s.c = 1;                             s.d = 1;
}                                    }
```

- Q: Is there a race?



Object Layout Considerations (2)

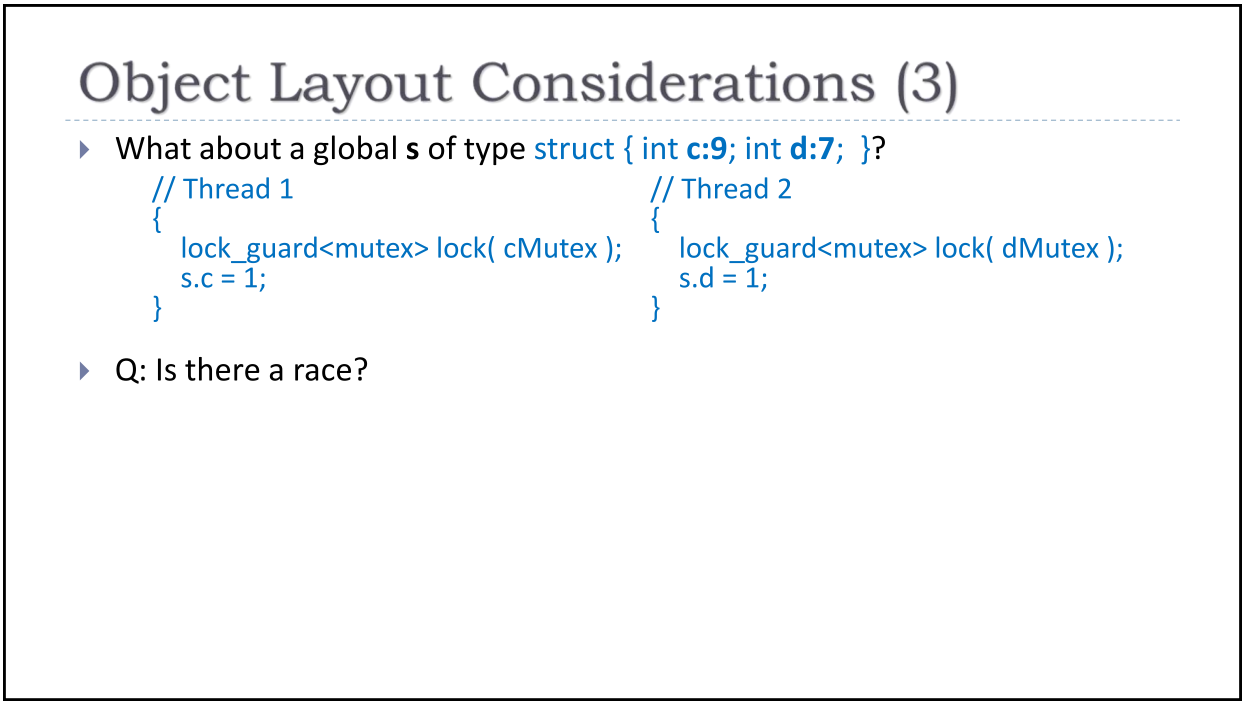
- What about a global **s** of type struct { char c; char d; }?

```
// Thread 1                             // Thread 2
{                                       {
   lock_guard<mutex> lock( cMutex );       lock_guard<mutex> lock( dMutex );
   s.c = 1;                                s.d = 1;
}                                       }
```

- Q: Is there a race? **No ideally and in C11/C++11, but maybe today:**
  - Say the system lays out *c* then *d* contiguously, and transforms "*d = 1*" to:

```
   char tmp[4];                         // 32-bit scratchpad
   memcpy( &tmp[0], &c, 4 );            // read 32 bits starting at c
   tmp[1] = 1;                          // set only the bits of d
   memcpy( &c, &tmp[0], 4 );            // write 32 bits back
```

  - Oops: Thread 2 would silently also write to *s.c* without holding *cMutex*.

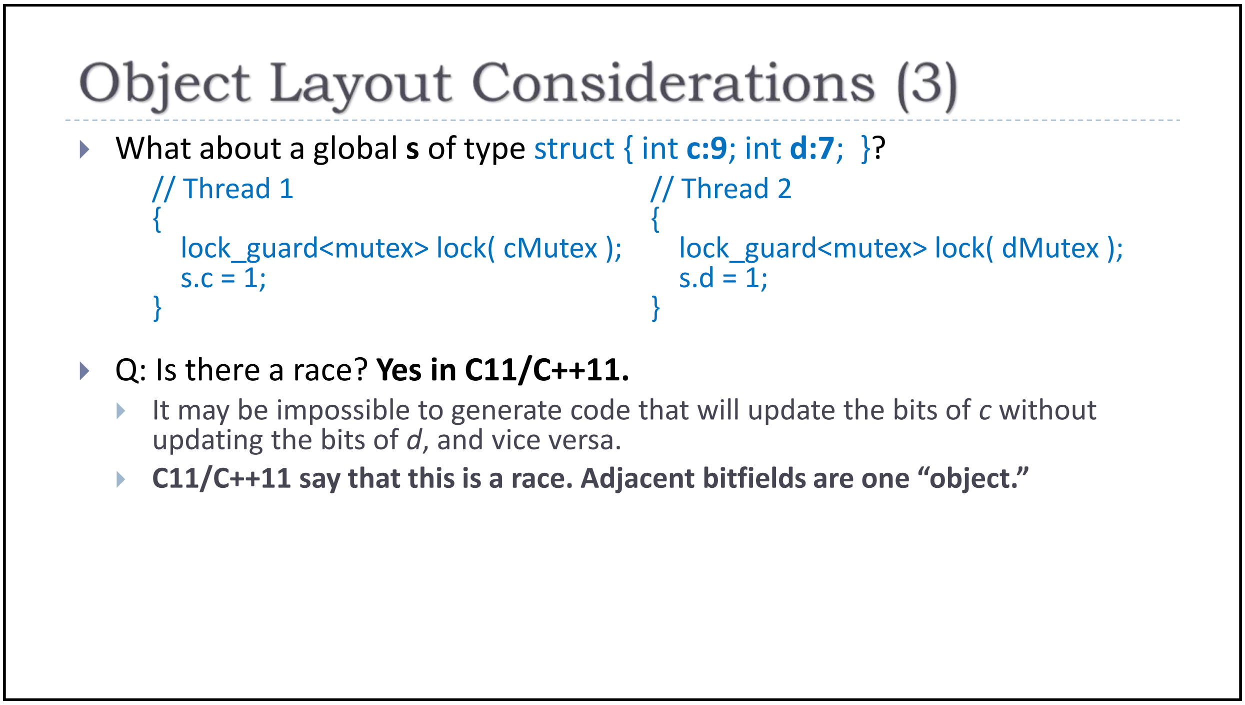


Object Layout Considerations (3)

- What about a global **s** of type struct { int **c:9**; int **d:7**; }?

```
// Thread 1                             // Thread 2
{                                       {
   lock_guard<mutex> lock( cMutex );       lock_guard<mutex> lock( dMutex );
   s.c = 1;                                s.d = 1;
}                                       }
```

- Q: Is there a race?

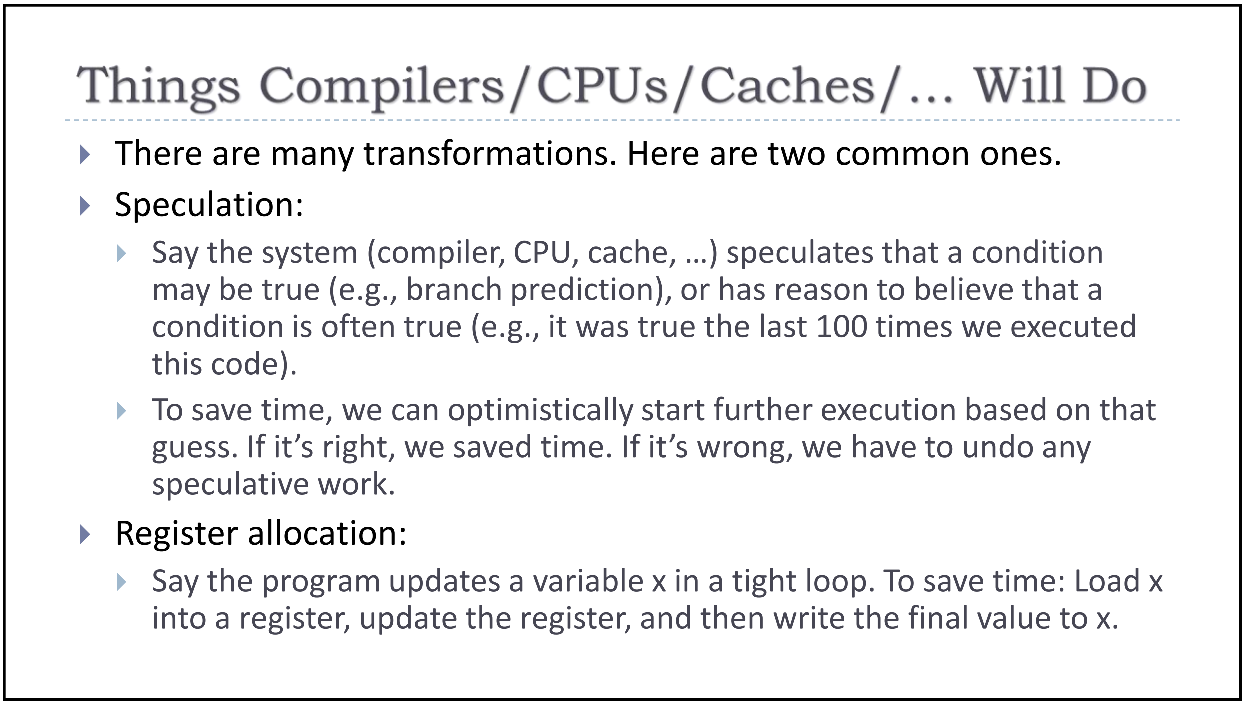


Object Layout Considerations (3)

- What about a global **s** of type struct { int **c:9**; int **d:7**; }?

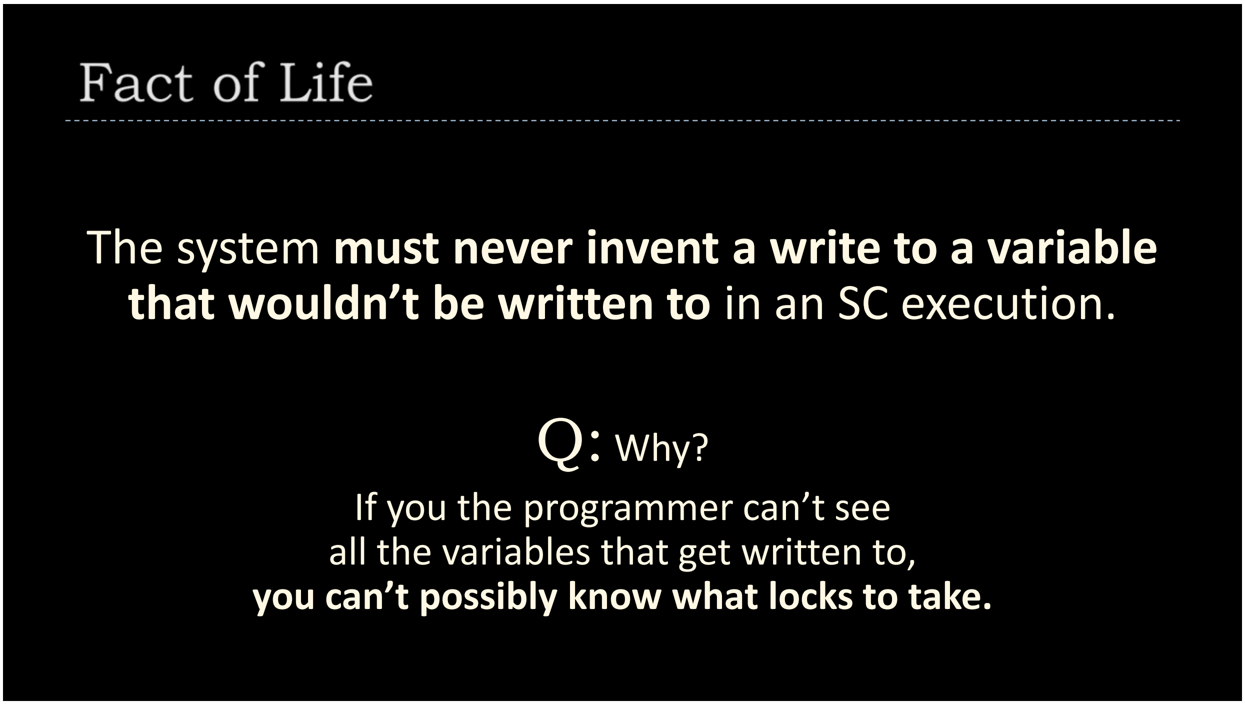
```
// Thread 1                                  // Thread 2
{                                            {
  lock_guard<mutex> lock( cMutex );            lock_guard<mutex> lock( dMutex );
  s.c = 1;                                     s.d = 1;
}                                            }
```

- Q: Is there a race? **Yes in C11/C++11.**
  - It may be impossible to generate code that will update the bits of *c* without updating the bits of *d*, and vice versa.
  - **C11/C++11 say that this is a race. Adjacent bitfields are one “object.”**



Things Compilers/CPUs/Caches/… Will Do

- There are many transformations. Here are two common ones.
- Speculation:
  - Say the system (compiler, CPU, cache, …) speculates that a condition may be true (e.g., branch prediction), or has reason to believe that a condition is often true (e.g., it was true the last 100 times we executed this code).
  - To save time, we can optimistically start further execution based on that guess. If it’s right, we saved time. If it’s wrong, we have to undo any speculative work.
- Register allocation:
  - Say the program updates a variable x in a tight loop. To save time: Load x into a register, update the register, and then write the final value to x.

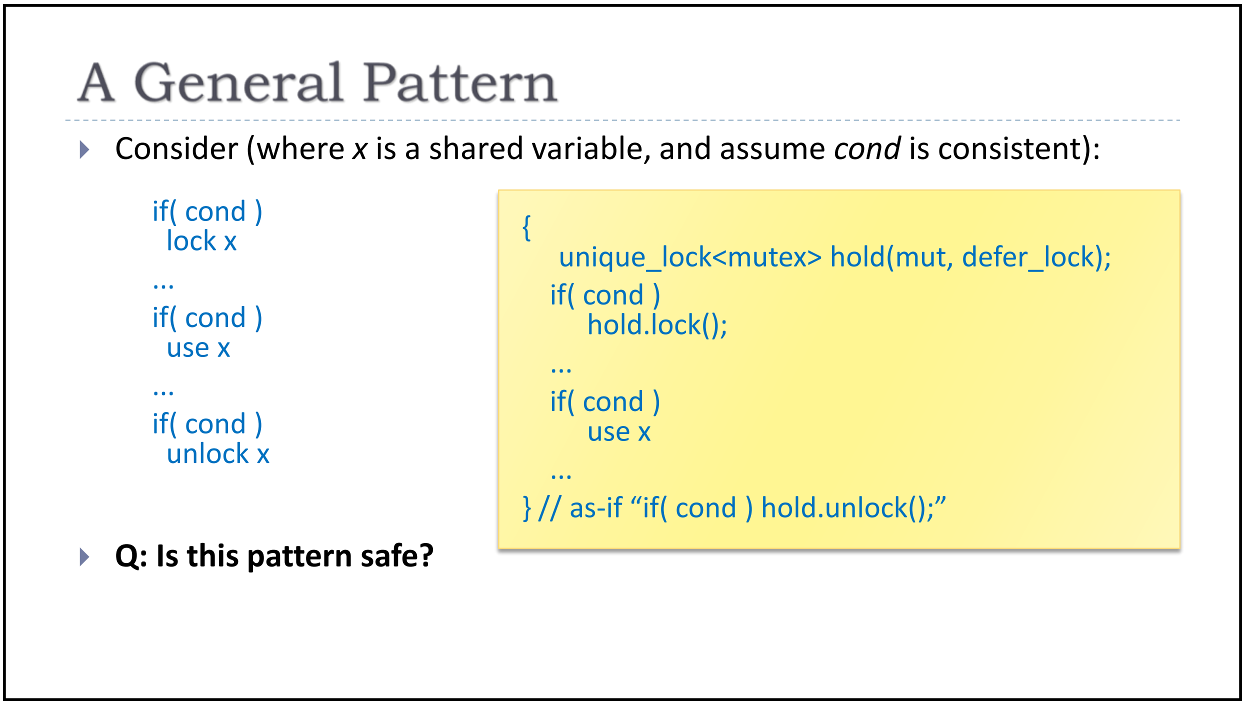


Fact of Life

The system **must never invent a write to a variable that wouldn't be written to** in an SC execution.

Q: Why?

If you the programmer can't see
all the variables that get written to,
you can't possibly know what locks to take.



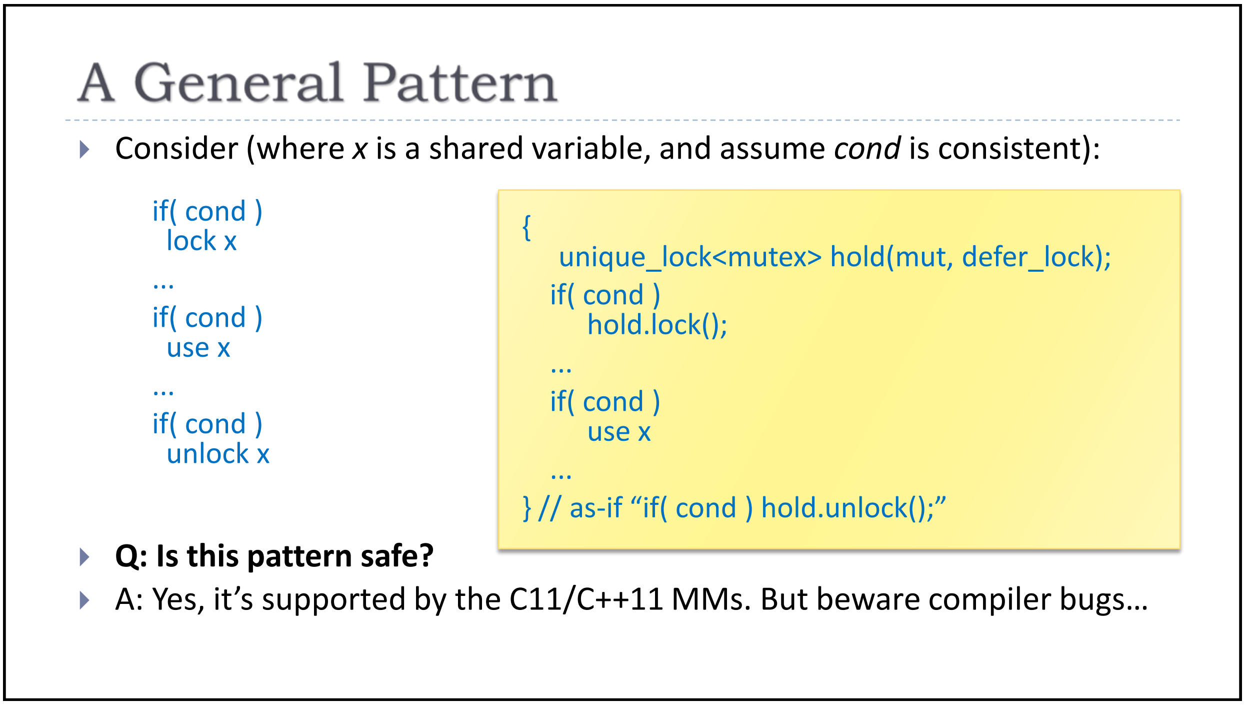
A General Pattern

- Consider (where *x* is a shared variable, and assume *cond* is consistent):

```
if( cond )
  lock x
…
if( cond )
  use x
…
if( cond )
  unlock x
```

```
{
   unique_lock<mutex> hold(mut, defer_lock);
  if( cond )
     hold.lock();
  …
  if( cond )
     use x
  …
} // as-if "if( cond ) hold.unlock();"
```

- **Q: Is this pattern safe?**



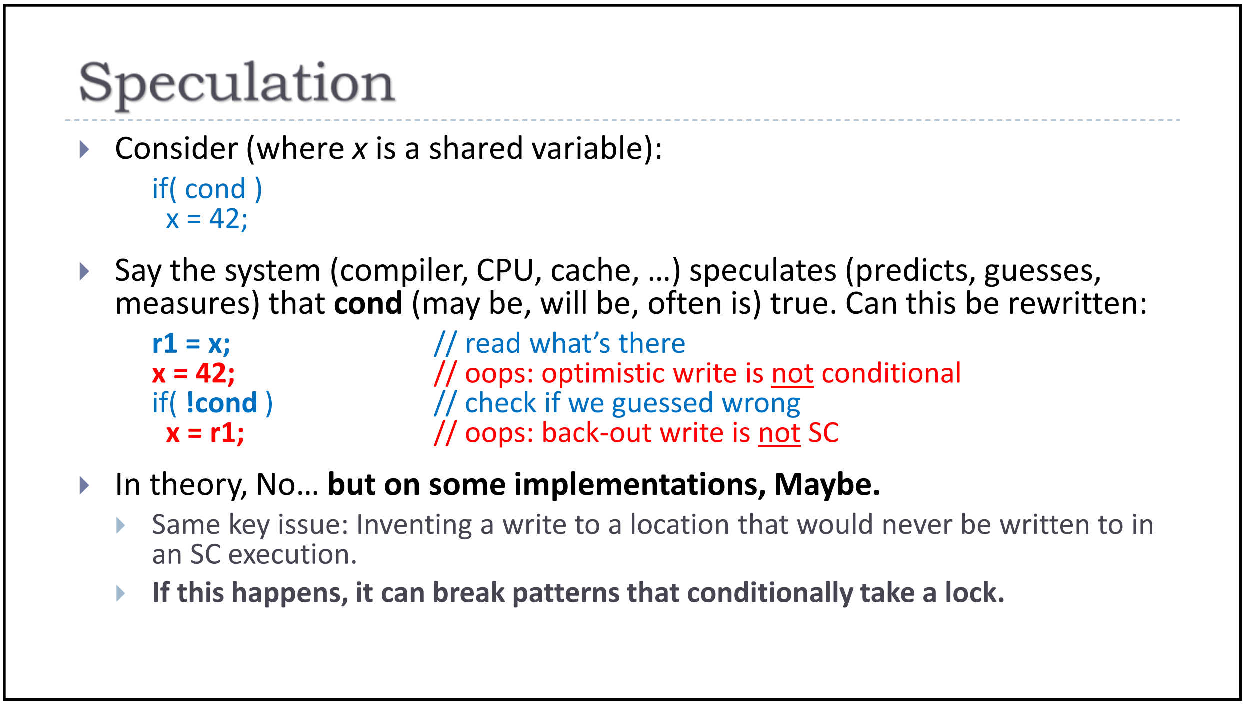
A General Pattern

- Consider (where *x* is a shared variable, and assume *cond* is consistent):

```
if( cond )
  lock x
…
if( cond )
  use x
…
if( cond )
  unlock x
```

```
{
  unique_lock<mutex> hold(mut, defer_lock);
  if( cond )
    hold.lock();
  …
  if( cond )
    use x
  …
} // as-if "if( cond ) hold.unlock();"
```

- **Q: Is this pattern safe?**
- A: Yes, it's supported by the C11/C++11 MMs. But beware compiler bugs…



Speculation

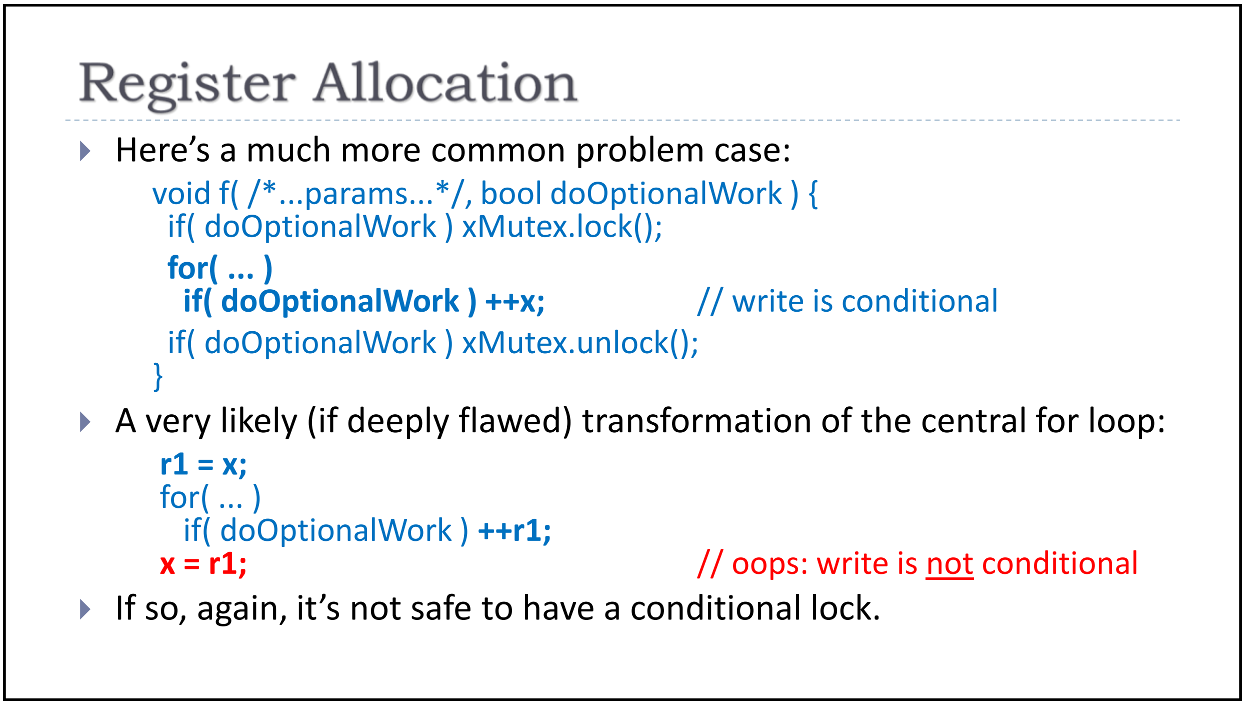
- Consider (where *x* is a shared variable):

```
if( cond )
  x = 42;
```

- Say the system (compiler, CPU, cache, …) speculates (predicts, guesses, measures) that **cond** (may be, will be, often is) true. Can this be rewritten:

```
r1 = x;            // read what's there
x = 42;            // oops: optimistic write is not conditional
if( !cond )        // check if we guessed wrong
  x = r1;          // oops: back-out write is not SC
```

- In theory, No… **but on some implementations, Maybe.**
  - Same key issue: Inventing a write to a location that would never be written to in an SC execution.
  - **If this happens, it can break patterns that conditionally take a lock.**



Register Allocation

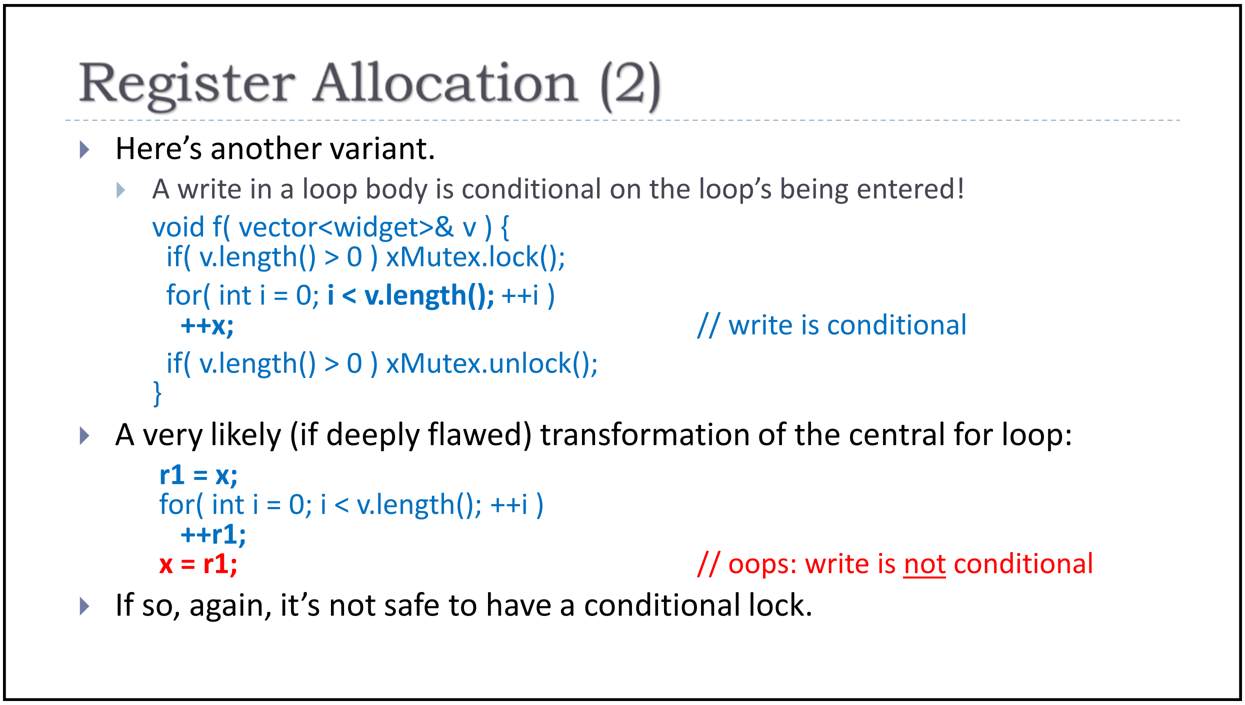
- Here’s a much more common problem case:

```
void f( /*...params...*/, bool doOptionalWork ) {
  if( doOptionalWork ) xMutex.lock();
  for( ... )
    if( doOptionalWork ) ++x;            // write is conditional
  if( doOptionalWork ) xMutex.unlock();
}
```

- A very likely (if deeply flawed) transformation of the central for loop:

```
r1 = x;
for( ... )
  if( doOptionalWork ) ++r1;
x = r1;                                  // oops: write is not conditional
```

- If so, again, it’s not safe to have a conditional lock.



Register Allocation (2)

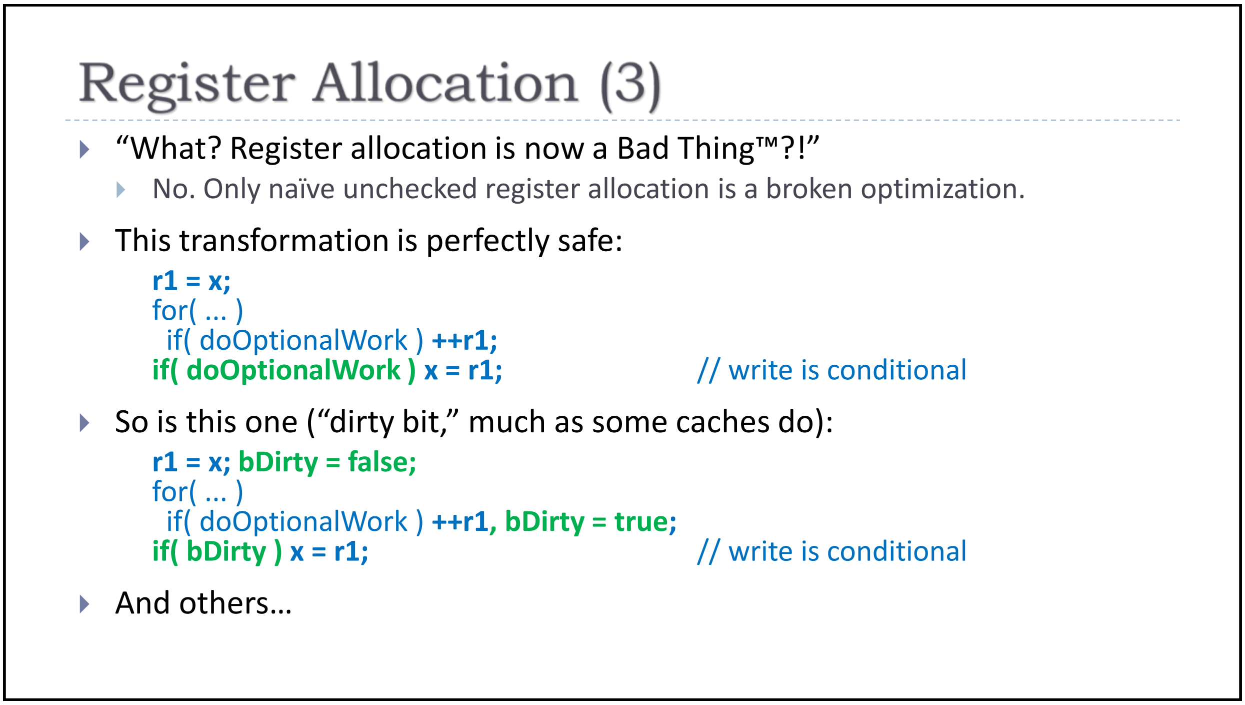
- Here’s another variant.
  - A write in a loop body is conditional on the loop’s being entered!

```
void f( vector<widget>& v ) {
  if( v.length() > 0 ) xMutex.lock();
  for( int i = 0; i < v.length(); ++i )
    ++x;                                 // write is conditional
  if( v.length() > 0 ) xMutex.unlock();
}
```

- A very likely (if deeply flawed) transformation of the central for loop:

```
r1 = x;
for( int i = 0; i < v.length(); ++i )
  ++r1;
x = r1;                                  // oops: write is not conditional
```

- If so, again, it’s not safe to have a conditional lock.



Register Allocation (3)

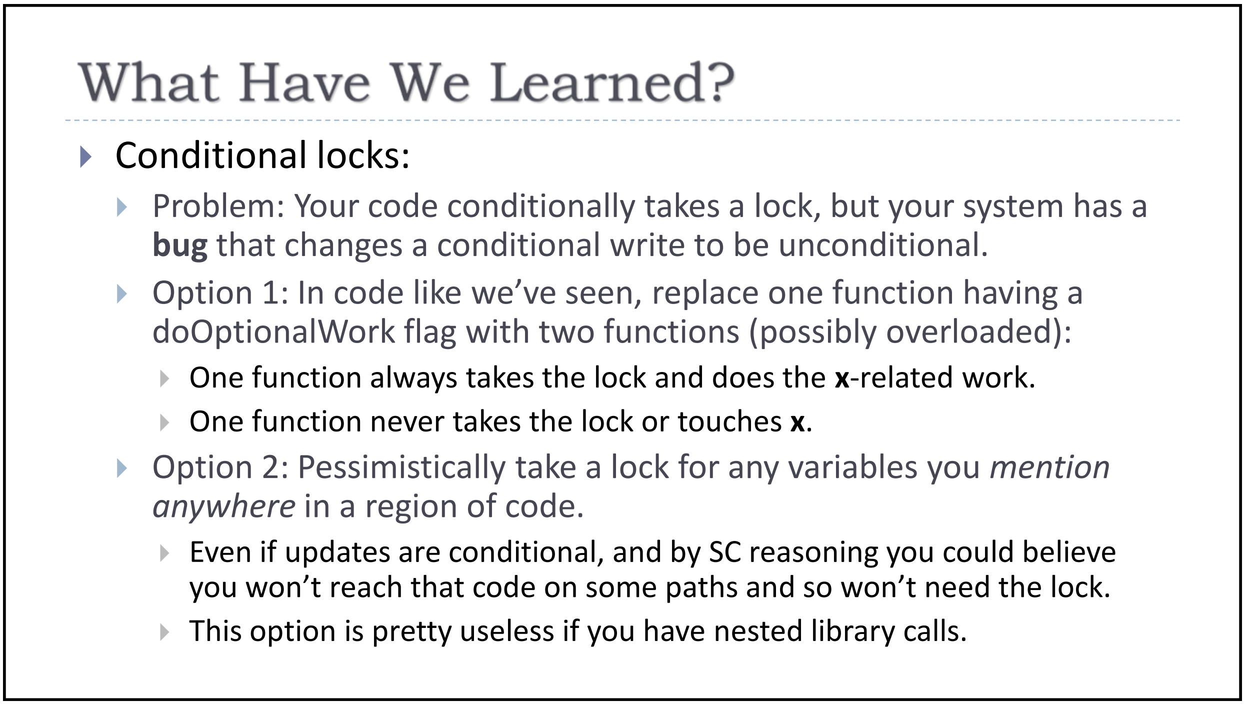
- “What? Register allocation is now a Bad Thing™?!”
  - No. Only naïve unchecked register allocation is a broken optimization.
- This transformation is perfectly safe:

```
r1 = x;
for( … )
  if( doOptionalWork ) ++r1;
if( doOptionalWork ) x = r1;                // write is conditional
```

- So is this one (“dirty bit,” much as some caches do):

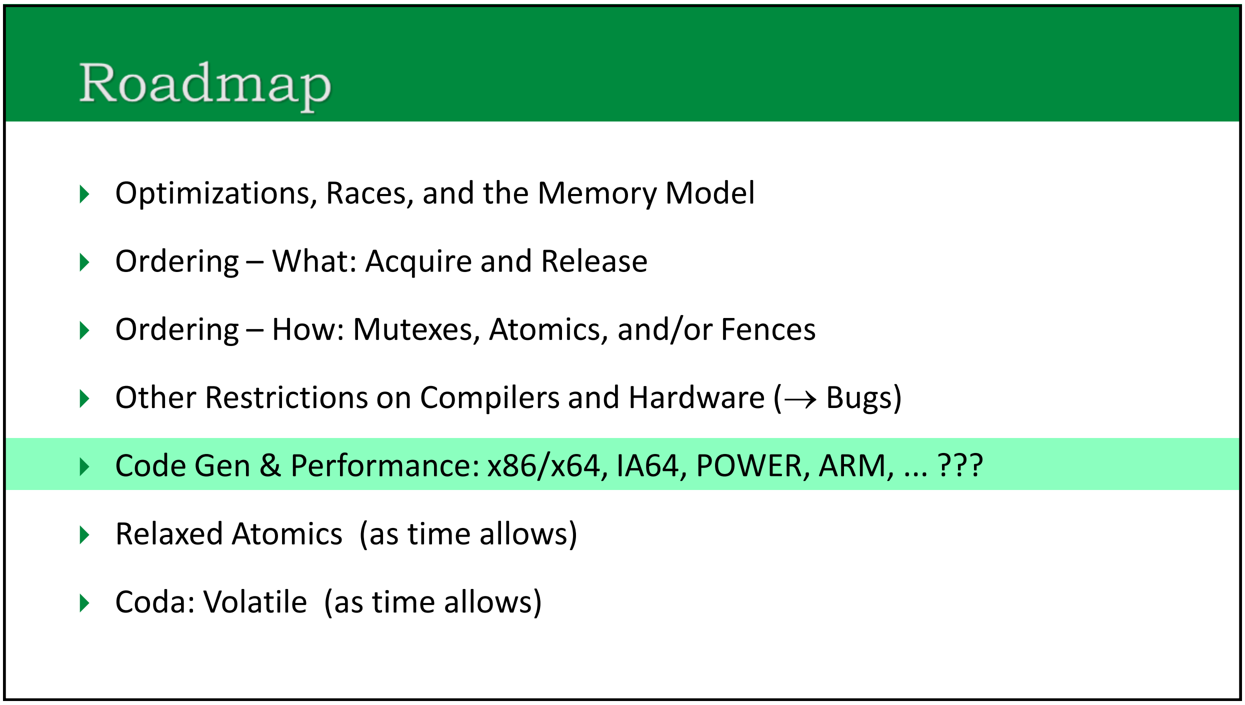
```
r1 = x; bDirty = false;
for( … )
  if( doOptionalWork ) ++r1, bDirty = true;
if( bDirty ) x = r1;                        // write is conditional
```

- And others…



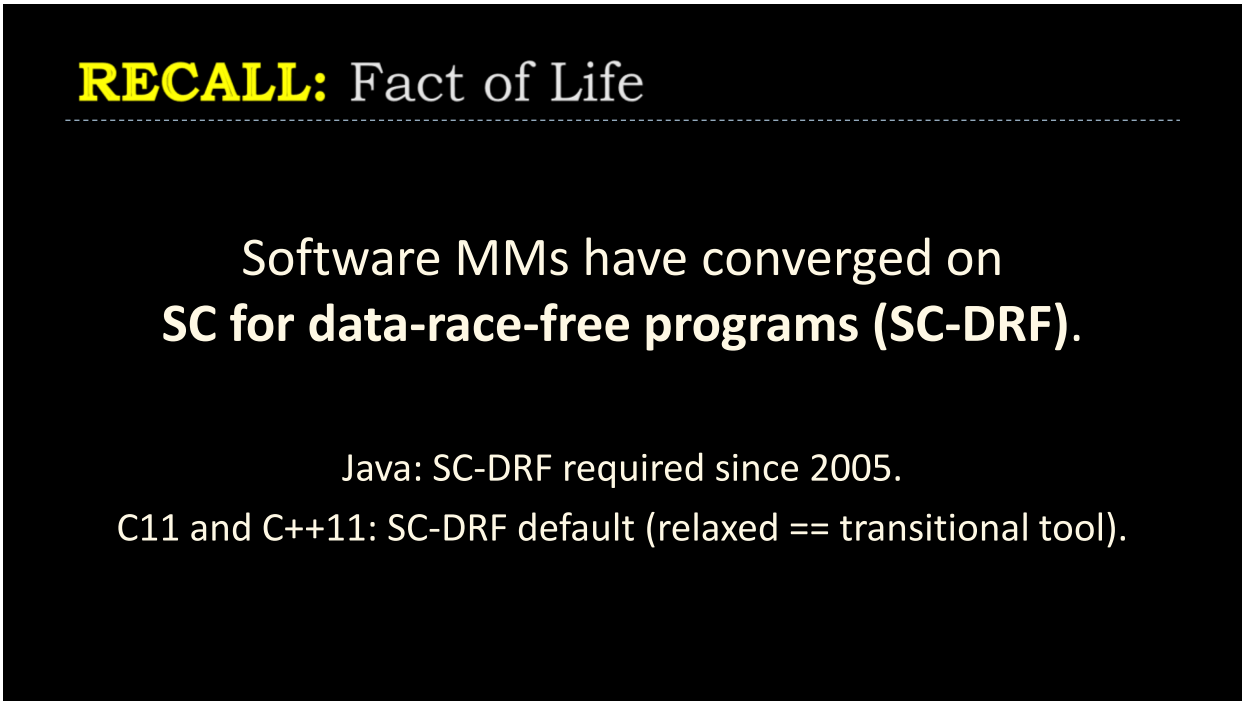
What Have We Learned?

- Conditional locks:
  - Problem: Your code conditionally takes a lock, but your system has a **bug** that changes a conditional write to be unconditional.
  - Option 1: In code like we’ve seen, replace one function having a doOptionalWork flag with two functions (possibly overloaded):
    - One function always takes the lock and does the **x**-related work.
    - One function never takes the lock or touches **x**.
  - Option 2: Pessimistically take a lock for any variables you *mention anywhere* in a region of code.
    - Even if updates are conditional, and by SC reasoning you could believe you won’t reach that code on some paths and so won’t need the lock.
    - This option is pretty useless if you have nested library calls.



Roadmap

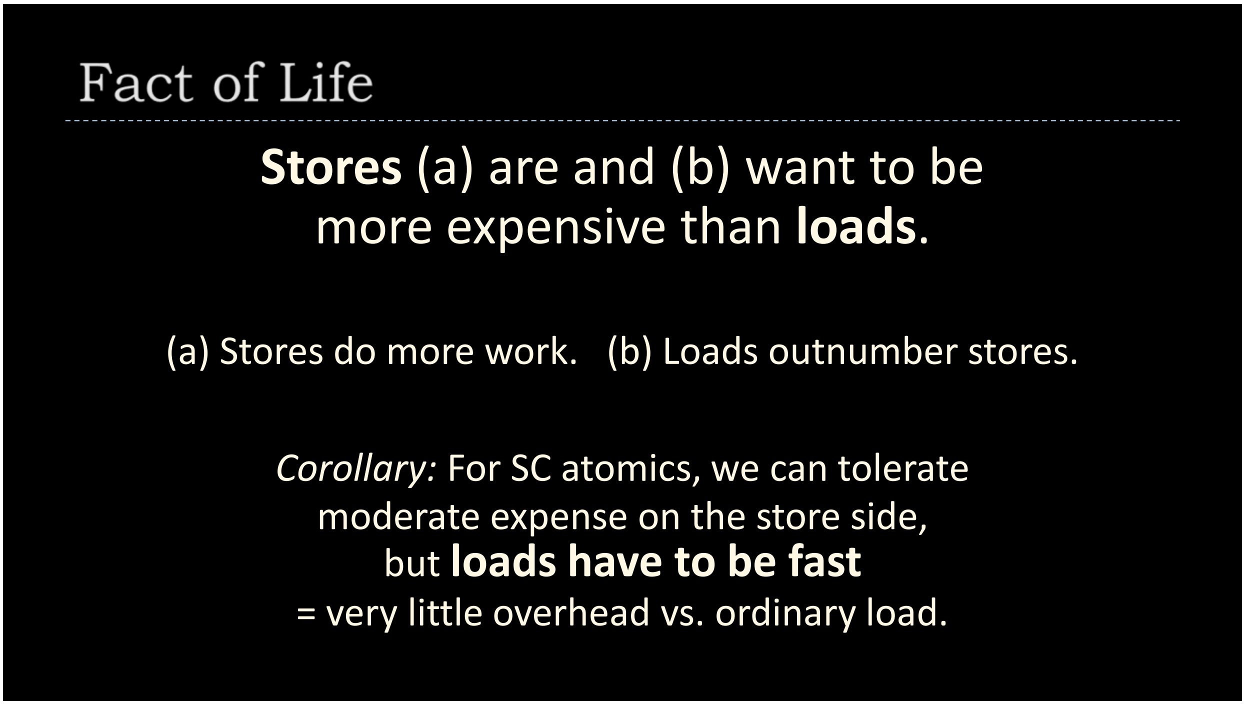
- Optimizations, Races, and the Memory Model
- Ordering – What: Acquire and Release
- Ordering – How: Mutexes, Atomics, and/or Fences
- Other Restrictions on Compilers and Hardware (→ Bugs)
- Code Gen & Performance: x86/x64, IA64, POWER, ARM, ... ???
- Relaxed Atomics (as time allows)
- Coda: Volatile (as time allows)



RECALL: Fact of Life

Software MMs have converged on
SC for data-race-free programs (SC-DRF).

Java: SC-DRF required since 2005.
C11 and C++11: SC-DRF default (relaxed == transitional tool).

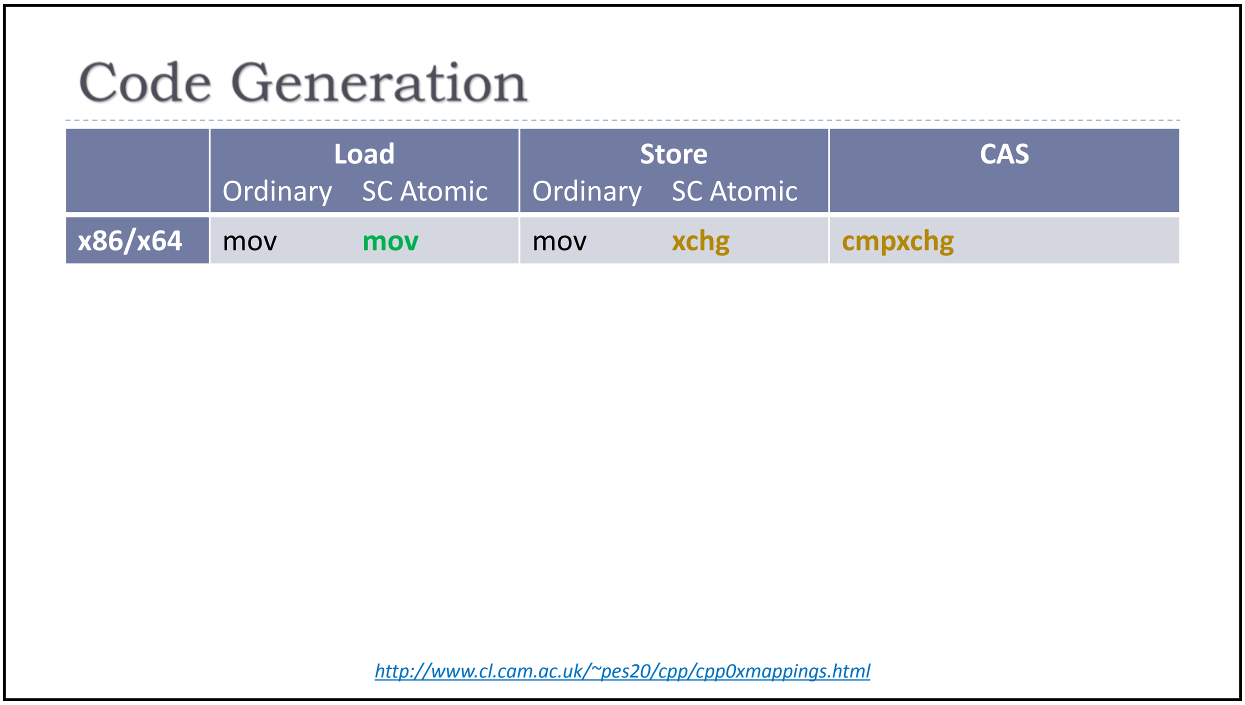


Fact of Life

Stores (a) are and (b) want to be more expensive than **loads.**

(a) Stores do more work. (b) Loads outnumber stores.

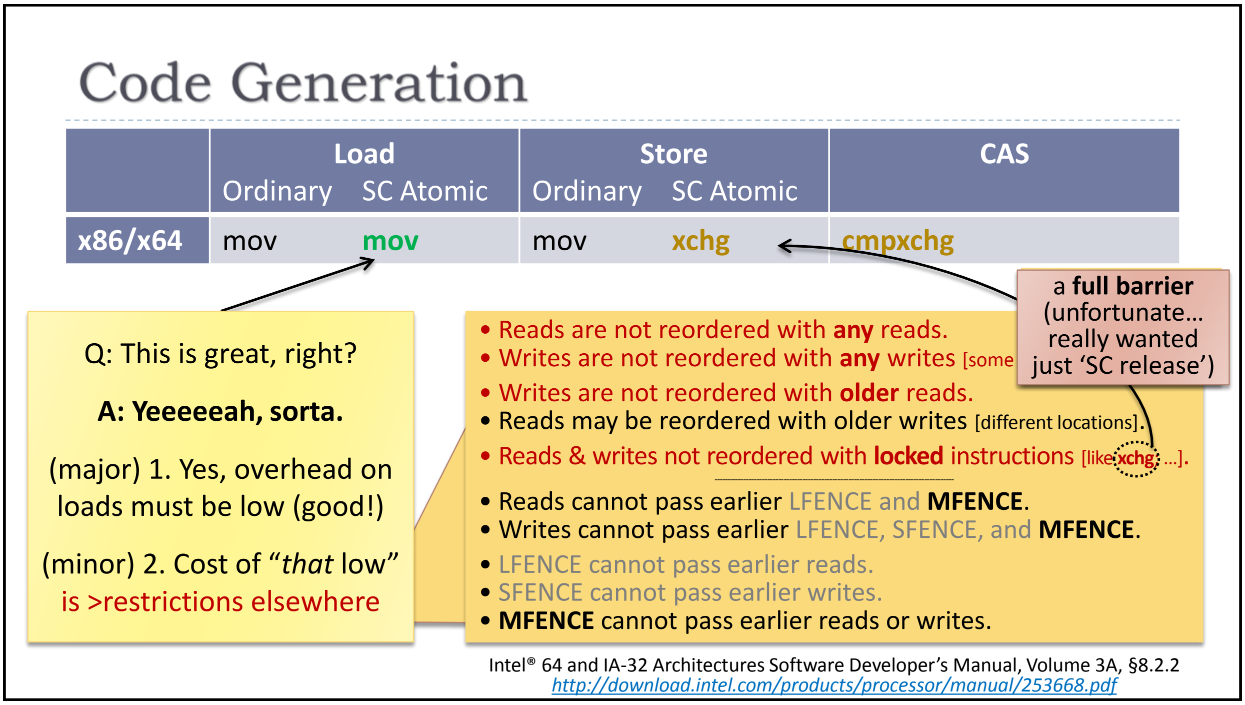
Corollary: For SC atomics, we can tolerate moderate expense on the store side, but **loads have to be fast** = very little overhead vs. ordinary load.



Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	**mov**	mov	**xchg**	**cmpxchg**

http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html



Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	mov	mov	xchg	cmpxchg

Q: This is great, right?

A: Yeeeeeah, sorta.

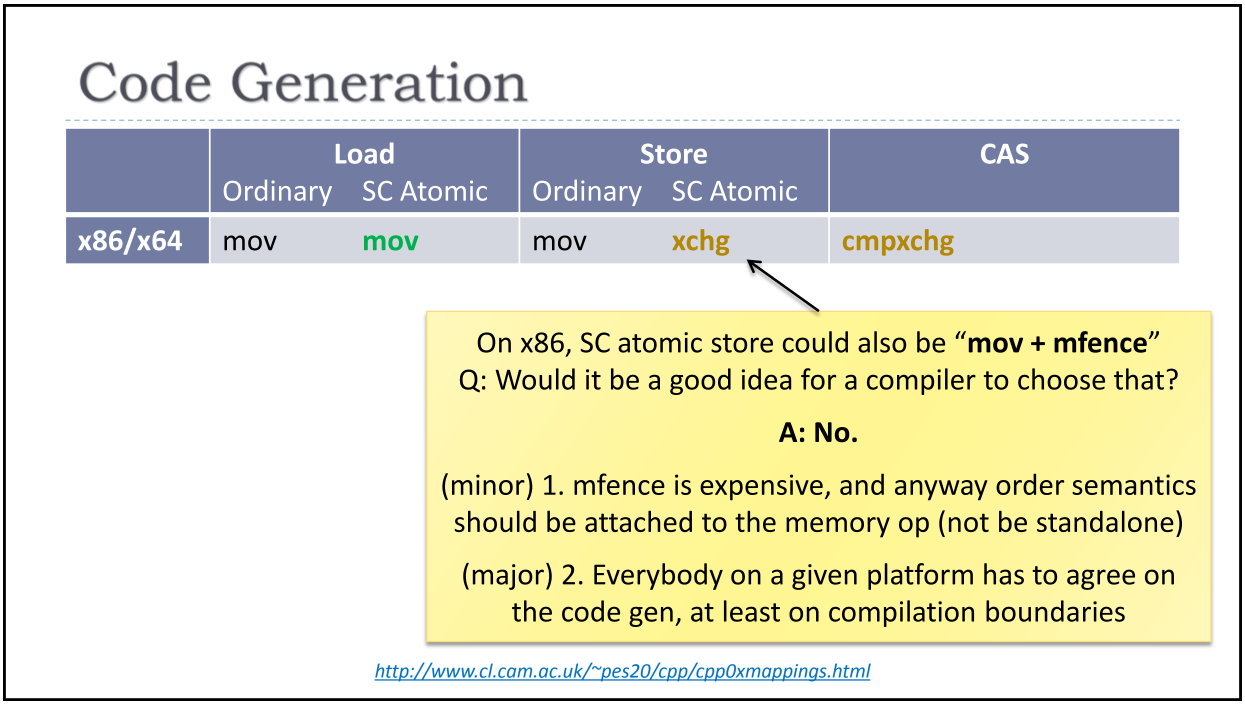
(major) 1. Yes, overhead on loads must be low (good!)

(minor) 2. Cost of "*that* low" is >restrictions elsewhere

a **full barrier** (unfortunate… really wanted just 'SC release')

- Reads are not reordered with **any** reads.
- Writes are not reordered with **any** writes [some
- Writes are not reordered with **older** reads.
- Reads may be reordered with older writes [different locations].
- Reads & writes not reordered with **locked** instructions [like xchg …].
- Reads cannot pass earlier LFENCE and **MFENCE**.
- Writes cannot pass earlier LFENCE, SFENCE, and **MFENCE**.
- LFENCE cannot pass earlier reads.
- SFENCE cannot pass earlier writes.
- **MFENCE** cannot pass earlier reads or writes.

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, §8.2.2
http://download.intel.com/products/processor/manual/253668.pdf



Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	mov	mov	xchg	cmpxchg

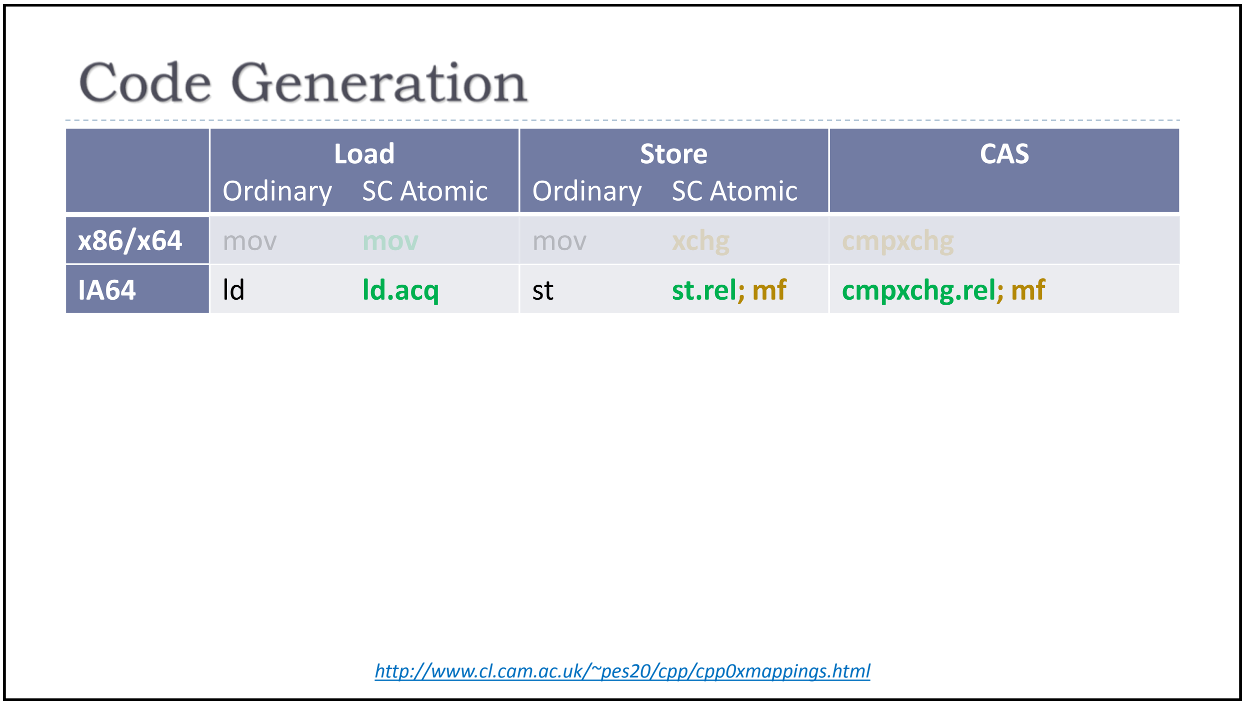
On x86, SC atomic store could also be "**mov + mfence**"
Q: Would it be a good idea for a compiler to choose that?

A: No.

(minor) 1. mfence is expensive, and anyway order semantics should be attached to the memory op (not be standalone)

(major) 2. Everybody on a given platform has to agree on the code gen, at least on compilation boundaries

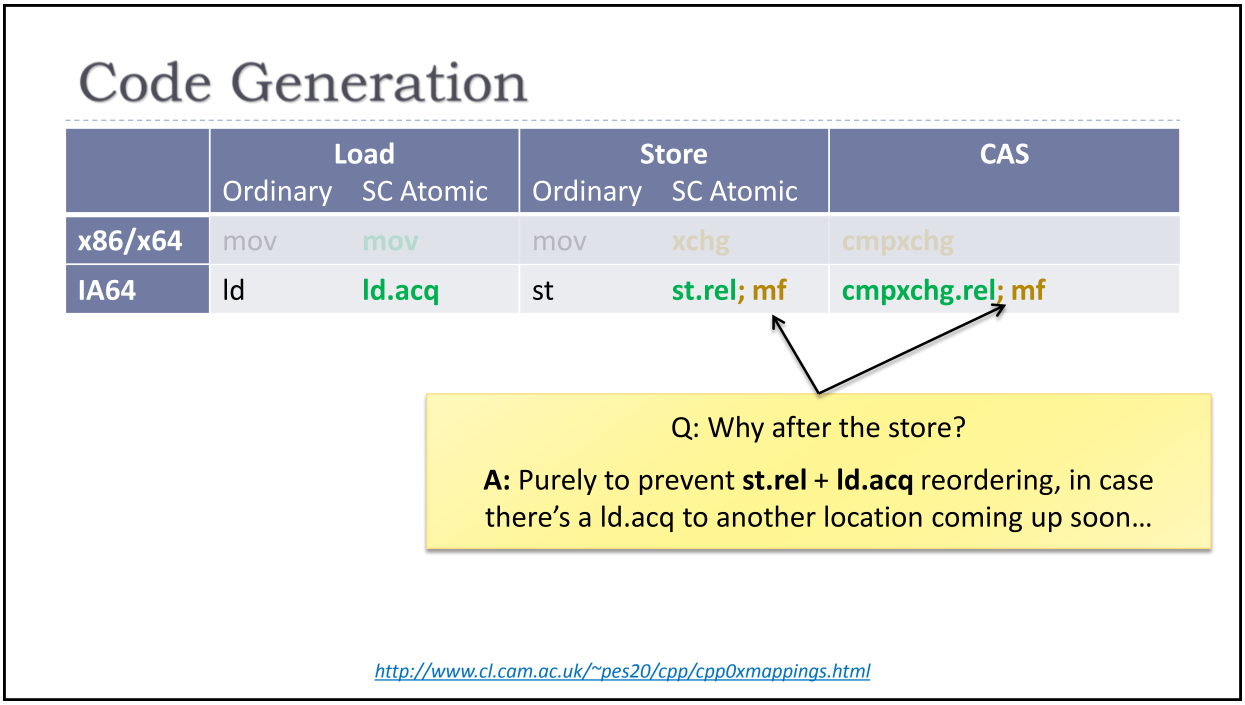
http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html



Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	mov	mov	xchg	cmpxchg
IA64	ld	**ld.acq**	st	**st.rel; mf**	**cmpxchg.rel; mf**

http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html



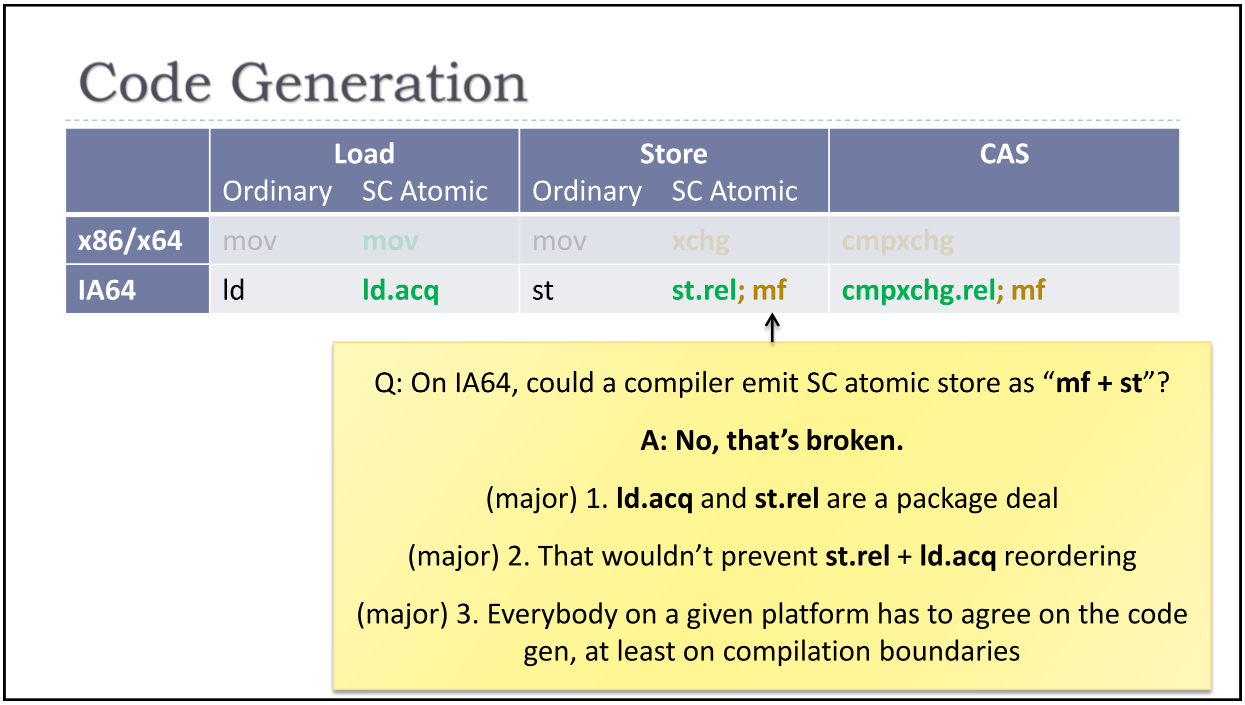
Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	mov	mov	xchg	cmpxchg
IA64	ld	**ld.acq**	st	**st.rel; mf**	**cmpxchg.rel; mf**

Q: Why after the store?

A: Purely to prevent **st.rel** + **ld.acq** reordering, in case there's a ld.acq to another location coming up soon…

http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html



Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	mov	mov	xchg	cmpxchg
IA64	ld	**ld.acq**	st	**st.rel; mf**	**cmpxchg.rel; mf**

↑

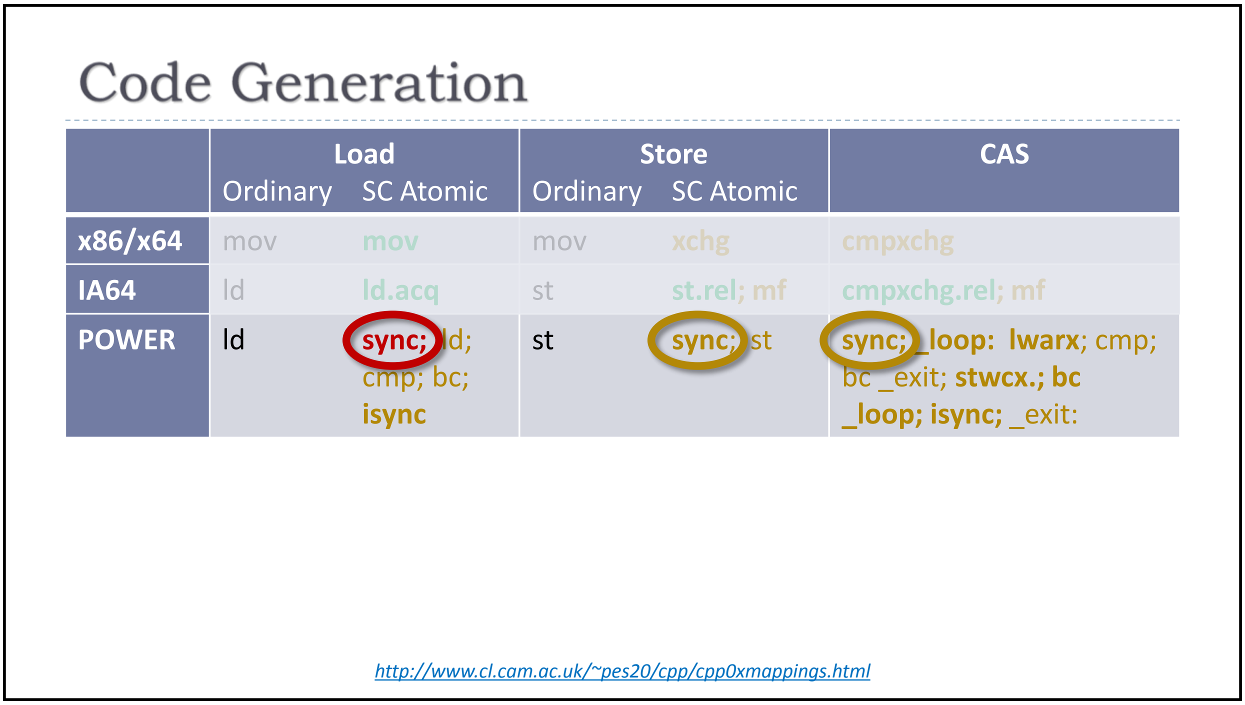
Q: On IA64, could a compiler emit SC atomic store as "**mf + st**"?

A: No, that's broken.

(major) 1. **ld.acq** and **st.rel** are a package deal

(major) 2. That wouldn't prevent **st.rel** + **ld.acq** reordering

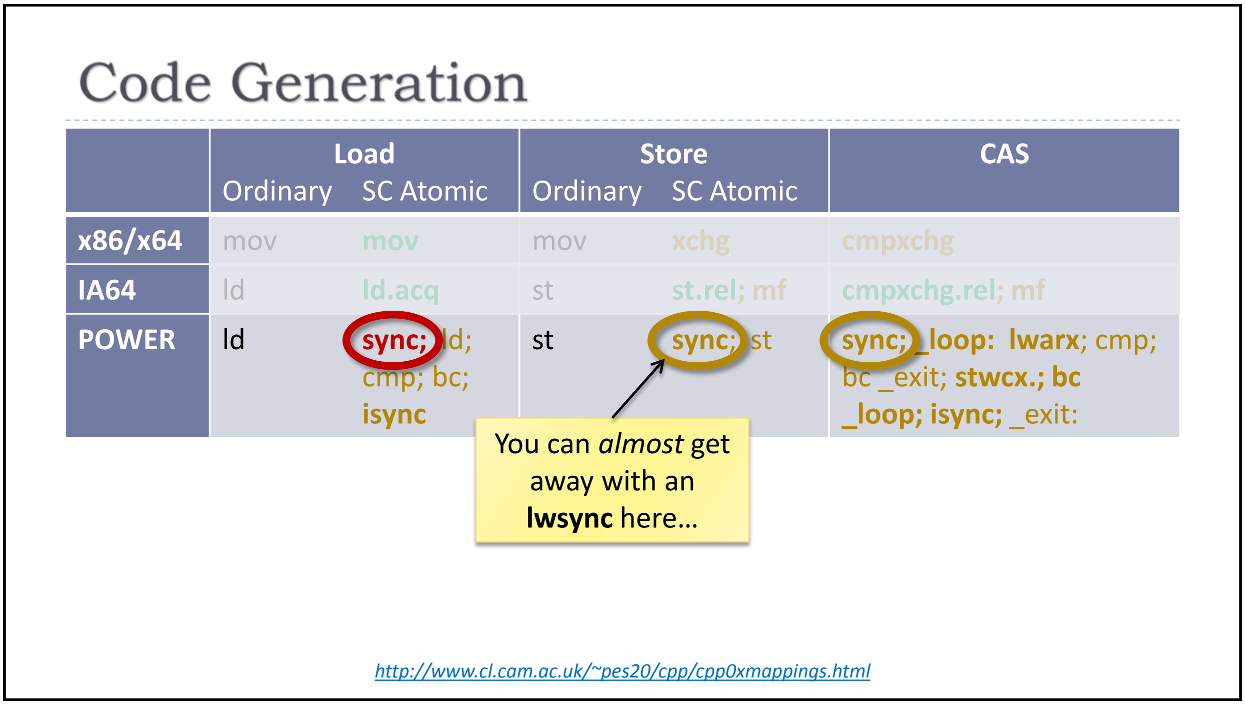
(major) 3. Everybody on a given platform has to agree on the code gen, at least on compilation boundaries



Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	mov	mov	xchg	cmpxchg
IA64	ld	ld.acq	st	st.rel; mf	cmpxchg.rel; mf
POWER	ld	**sync;** ld; cmp; bc; **isync**	st	**sync;** st	**sync; _loop: lwarx**; cmp; bc _exit; **stwcx.; bc _loop; isync;** _exit:

http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

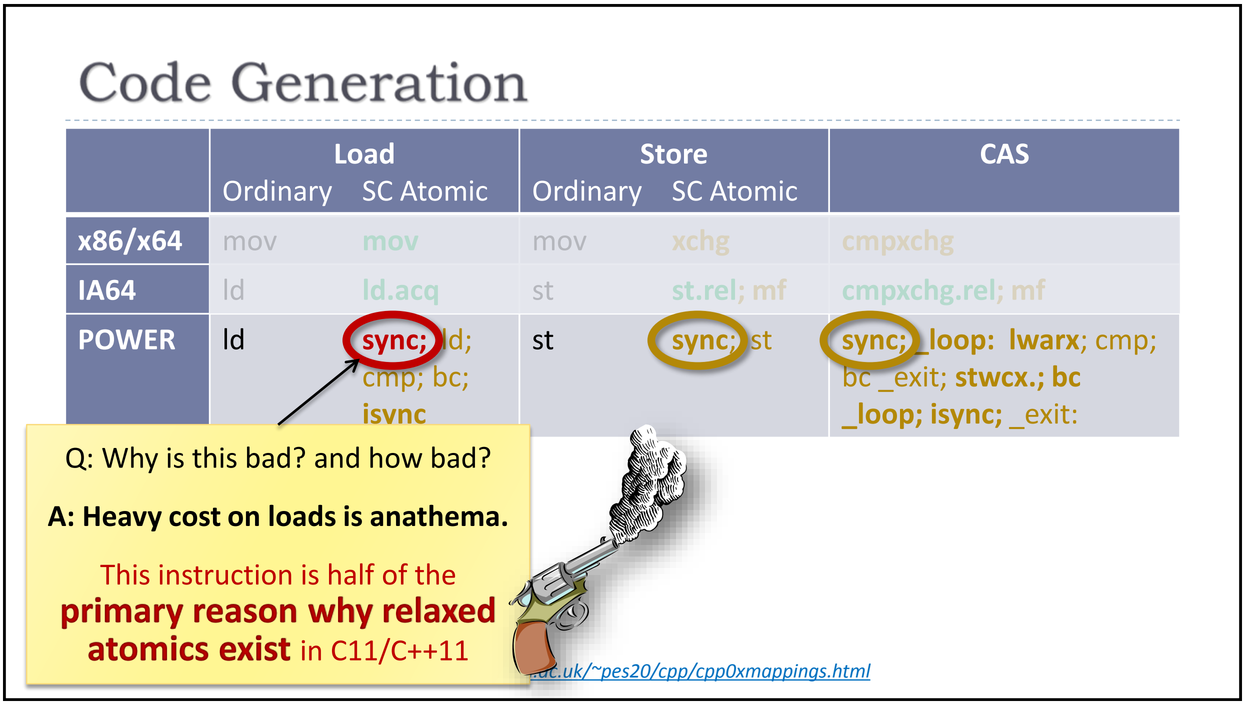


Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	mov	mov	xchg	cmpxchg
IA64	ld	ld.acq	st	st.rel; mf	cmpxchg.rel; mf
POWER	ld	**sync;** ld; cmp; bc; **isync**	st	**sync;** st	**sync;** **_loop: lwarx**; cmp; bc _exit; **stwcx.; bc _loop; isync;** _exit:

You can *almost* get away with an **lwsync** here…

http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html



Code Generation

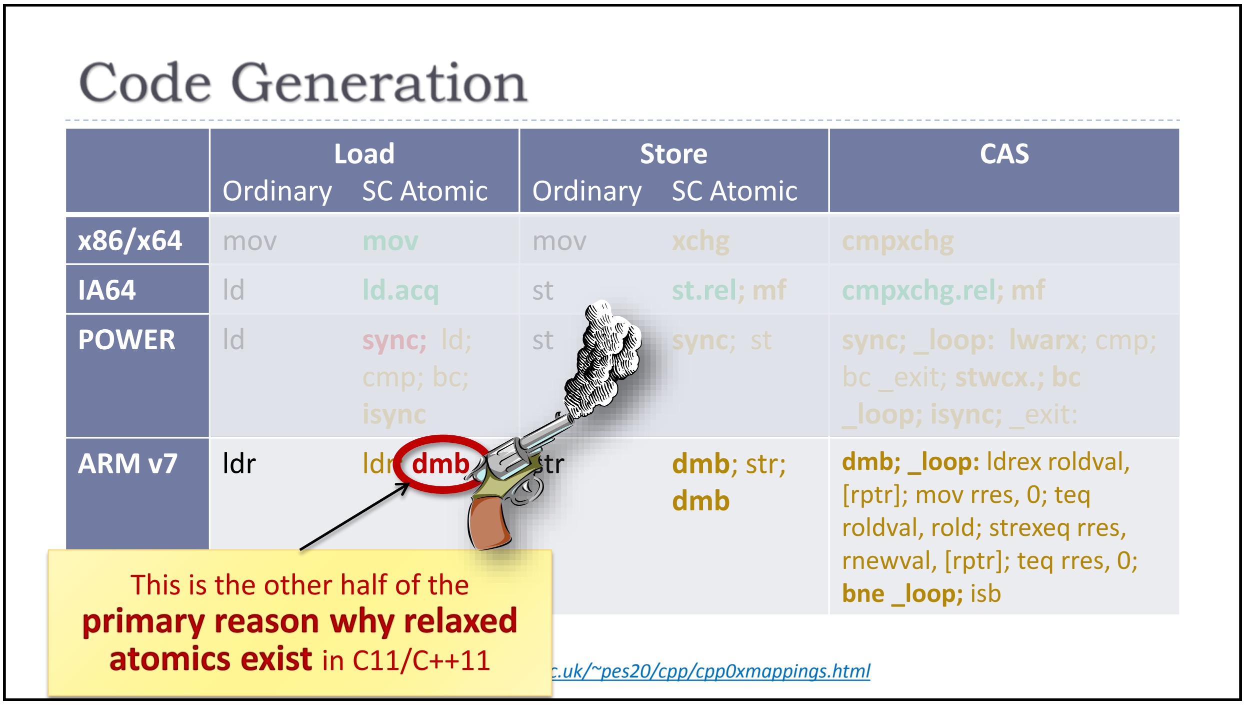
	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	mov	mov	xchg	cmpxchg
IA64	ld	ld.acq	st	st.rel; mf	cmpxchg.rel; mf
POWER	ld	**sync;** ld; cmp; bc; **isync**	st	**sync;** st	**sync;** **_loop: lwarx**; cmp; bc _exit; **stwcx.; bc _loop; isync;** _exit:

Q: Why is this bad? and how bad?

A: Heavy cost on loads is anathema.

This instruction is half of the **primary reason why relaxed atomics exist** in C11/C++11

ac.uk/~pes20/cpp/cpp0xmappings.html



Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	**mov**	mov	**xchg**	**cmpxchg**
IA64	ld	**ld.acq**	st	**st.rel; mf**	**cmpxchg.rel; mf**
POWER	ld	**sync;** ld; cmp; bc; **isync**	st	**sync;** st	**sync; _loop: lwarx;** cmp; bc _exit; **stwcx.; bc _loop; isync;** _exit:
ARM v7	ldr	ldr **dmb**	str	**dmb**; str; **dmb**	**dmb; _loop:** ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; **bne _loop;** isb

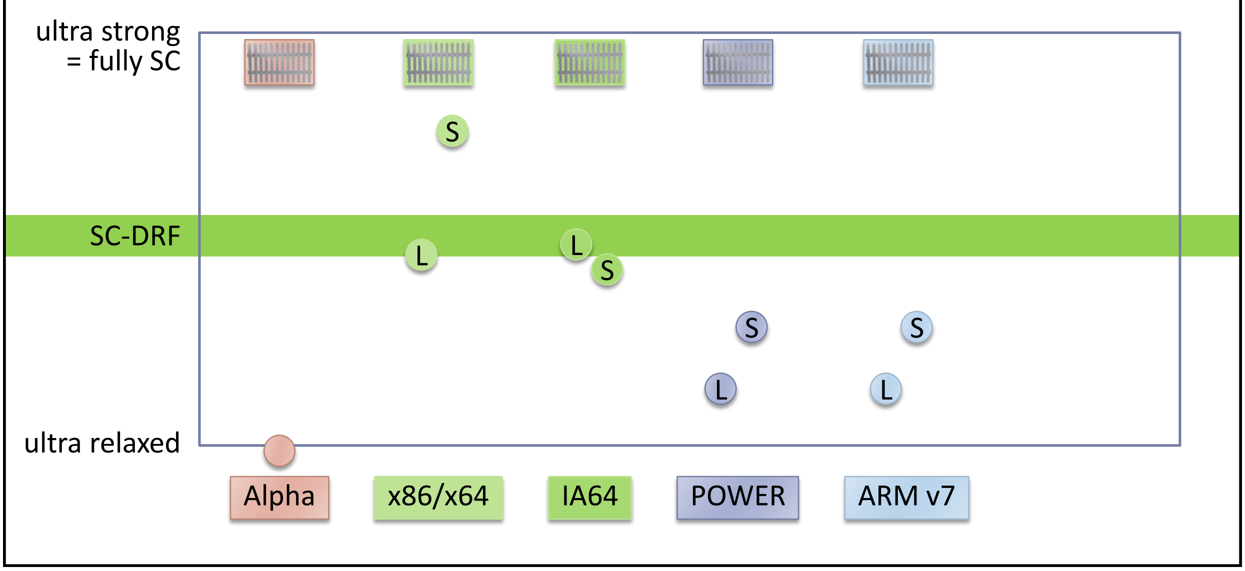
This is the other half of the **primary reason why relaxed atomics exist** in C11/C++11

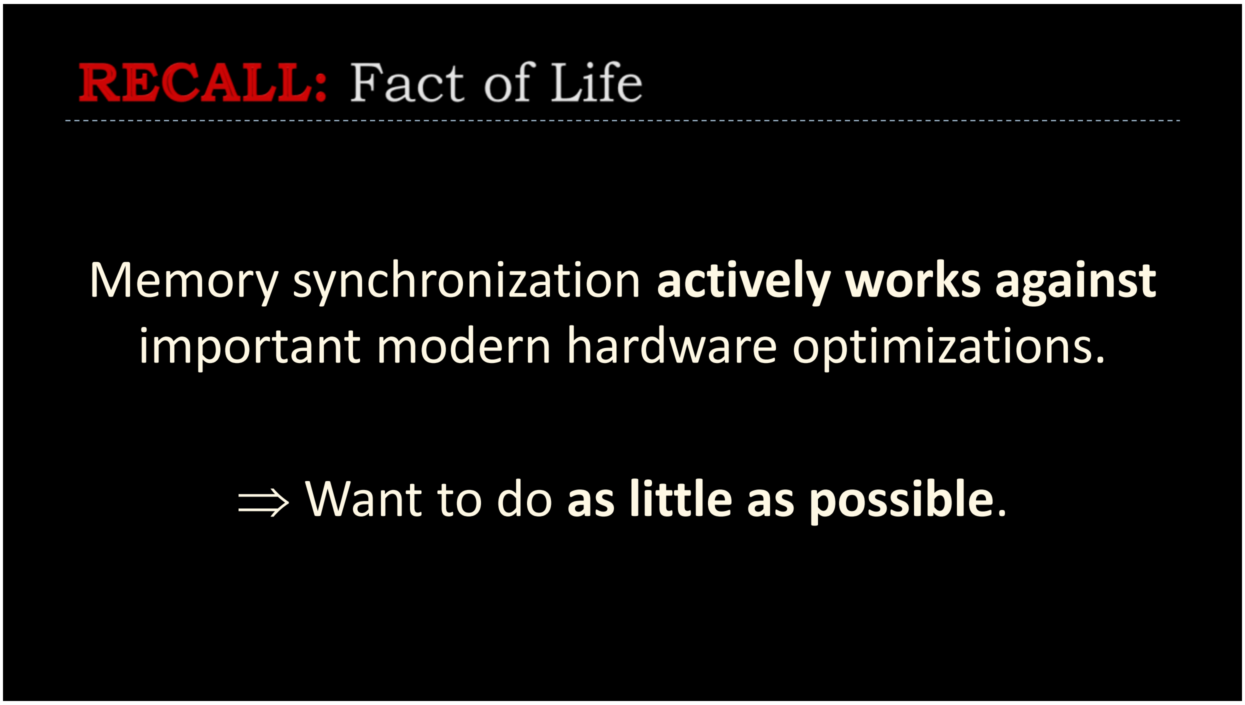
c.uk/~pes20/cpp/cpp0xmappings.html

On the Charts

○ loads and stores

▨ fences/barriers

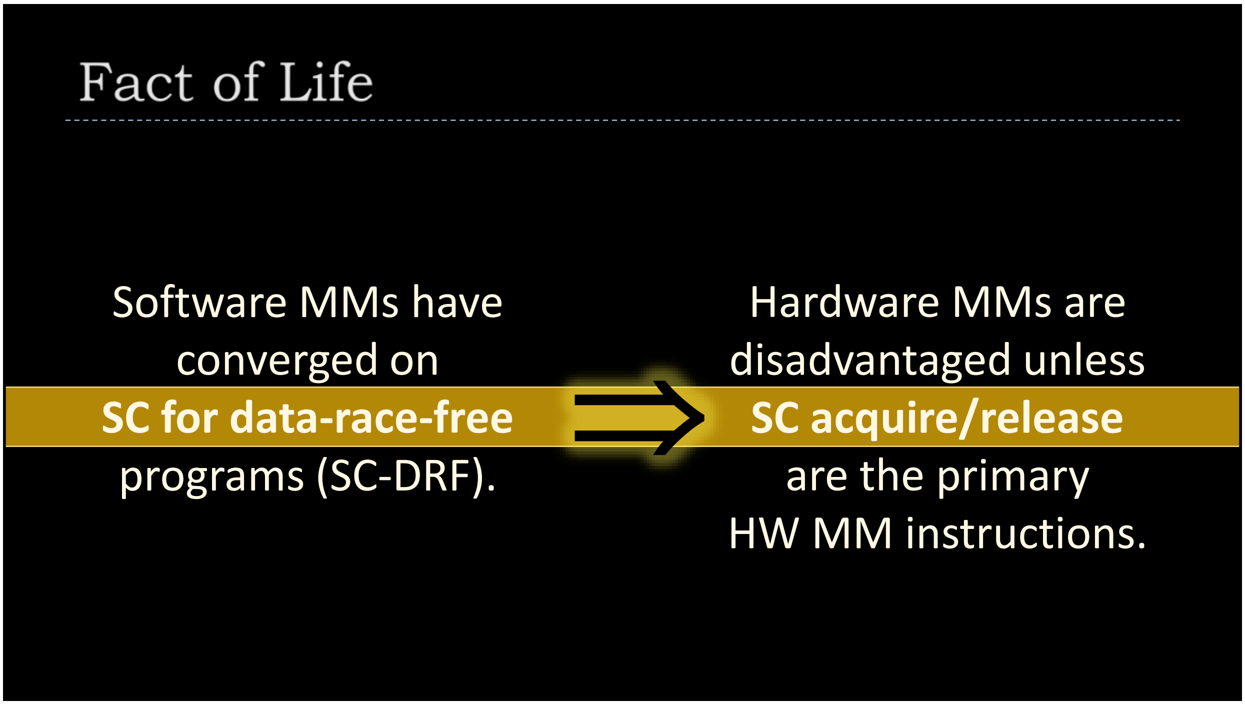




RECALL: Fact of Life

Memory synchronization **actively works against** important modern hardware optimizations.

⇒ Want to do **as little as possible**.

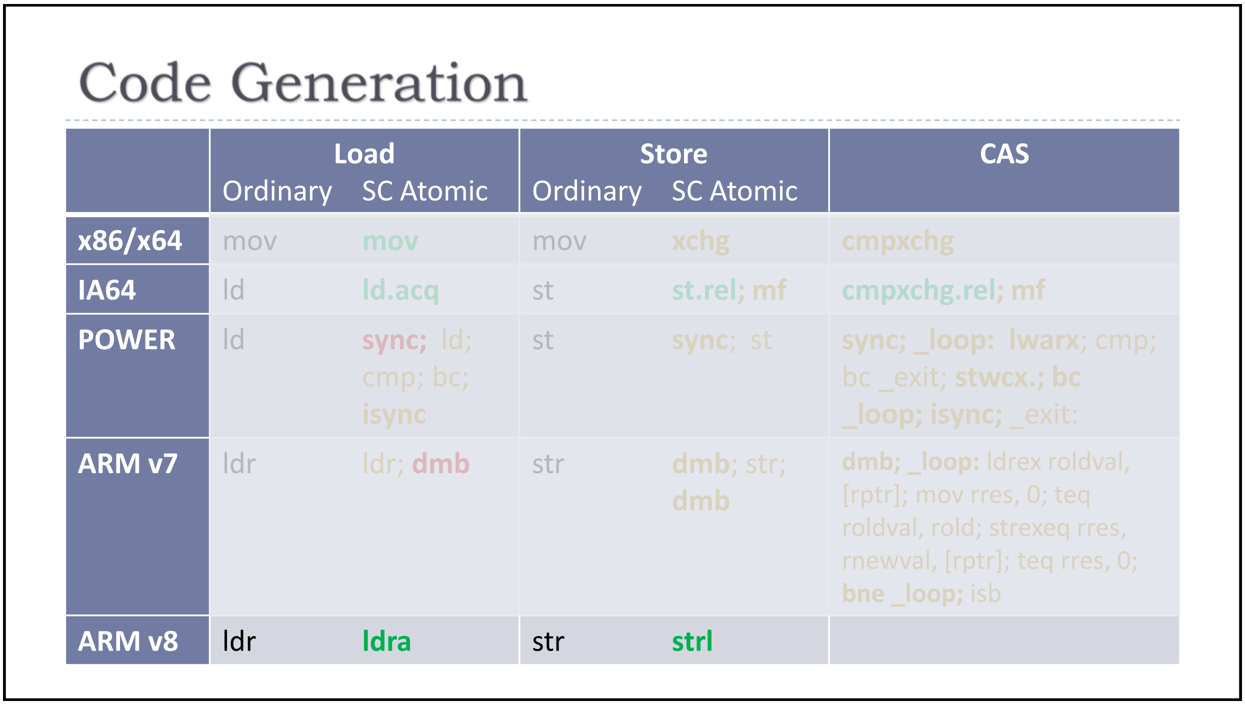


Fact of Life

Software MMs have converged on **SC for data-race-free** programs (SC-DRF).

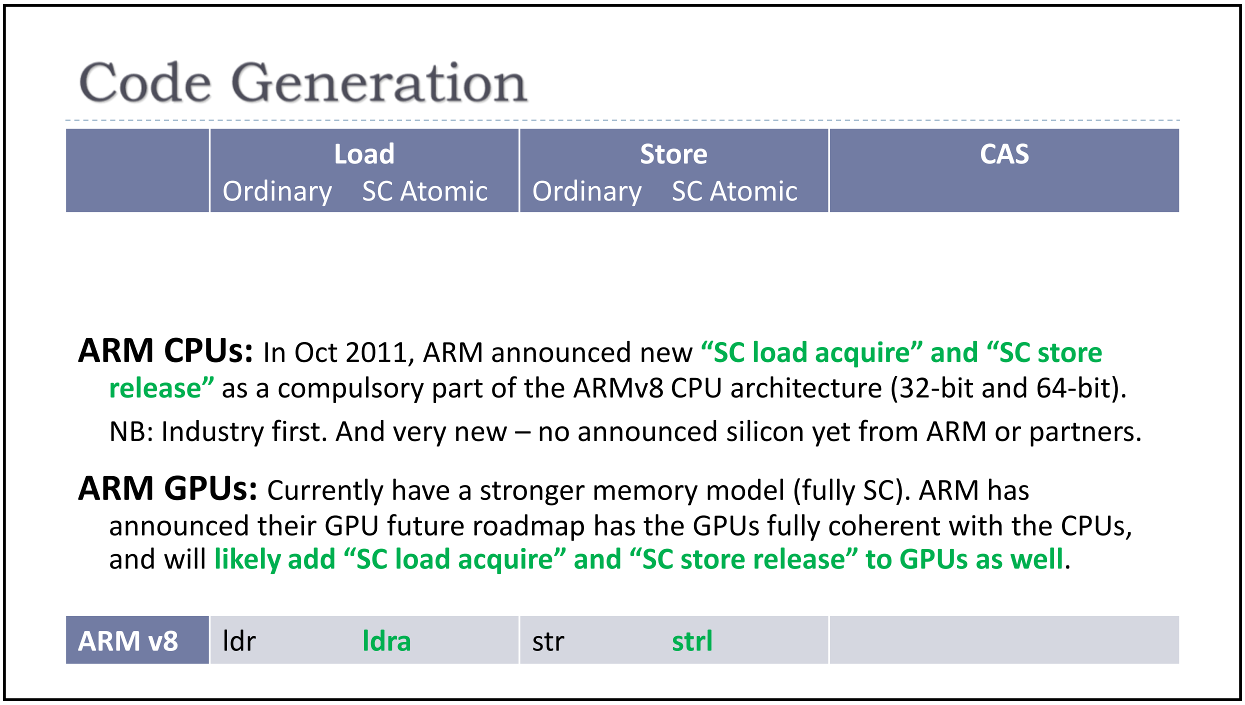
⇒

Hardware MMs are disadvantaged unless **SC acquire/release** are the primary HW MM instructions.



Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	mov	mov	xchg	cmpxchg
IA64	ld	ld.acq	st	st.rel; mf	cmpxchg.rel; mf
POWER	ld	sync; ld; cmp; bc; isync	st	sync; st	sync; _loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; isync; _exit:
ARM v7	ldr	ldr; dmb	str	dmb; str; dmb	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb
ARM v8	ldr	ldra	str	strl	



Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	

ARM CPUs: In Oct 2011, ARM announced new **"SC load acquire" and "SC store release"** as a compulsory part of the ARMv8 CPU architecture (32-bit and 64-bit).

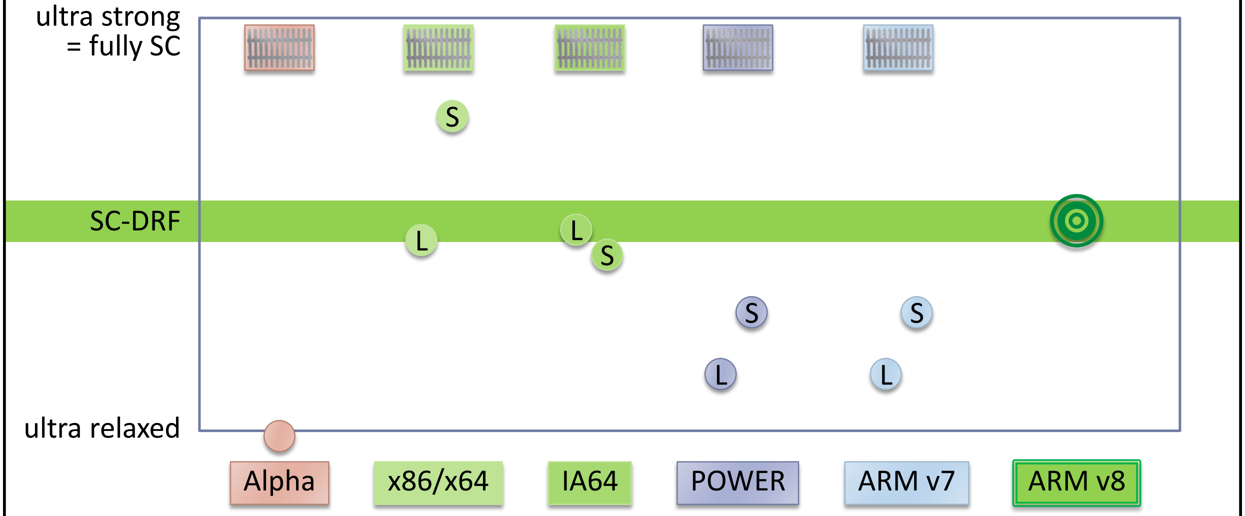
NB: Industry first. And very new – no announced silicon yet from ARM or partners.

ARM GPUs: Currently have a stronger memory model (fully SC). ARM has announced their GPU future roadmap has the GPUs fully coherent with the CPUs, and will **likely add "SC load acquire" and "SC store release" to GPUs as well**.

ARM v8	ldr	ldra	str	strl	

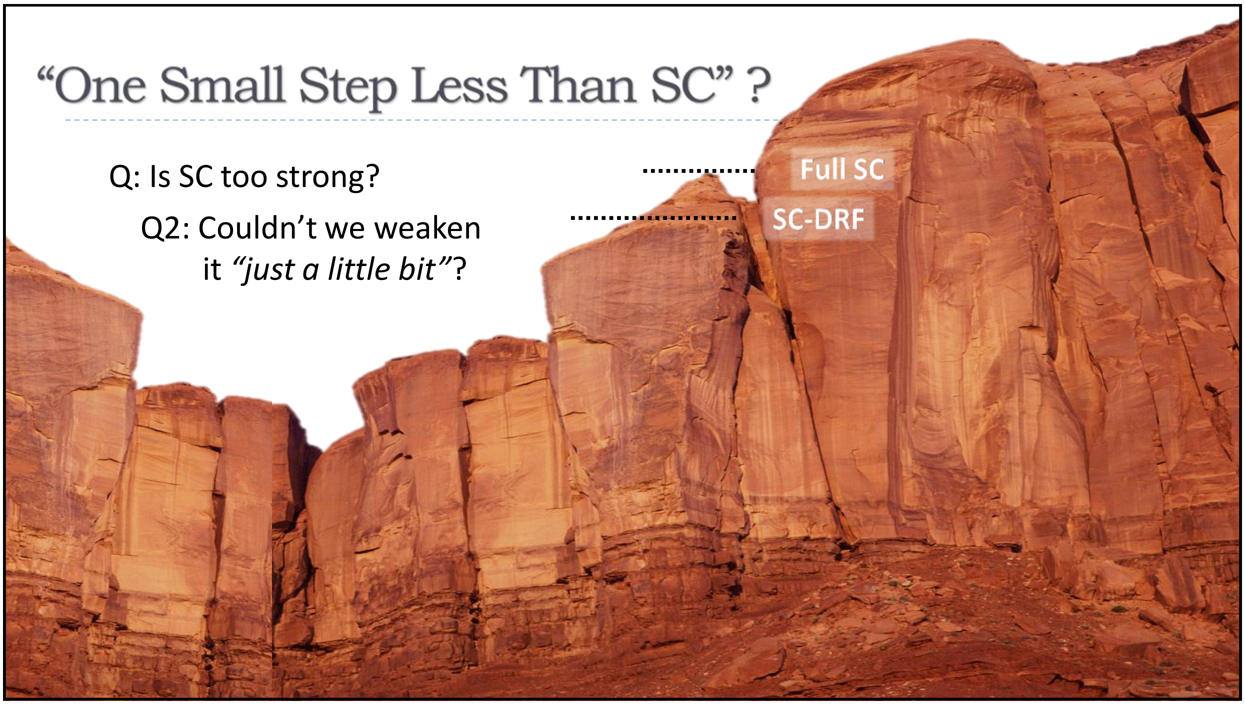
On the Charts

○ loads and stores
▨ fences/barriers



Roadmap

- Optimizations, Races, and the Memory Model
- Ordering – What: Acquire and Release
- Ordering – How: Mutexes, Atomics, and/or Fences
- Other Restrictions on Compilers and Hardware (→ Bugs)
- Code Gen & Performance: x86/x64, IA64, POWER, ARM, ... ???
- Relaxed Atomics (as time allows)
- Coda: Volatile (as time allows)



“One Small Step Less Than SC” ?

Q: Is SC too strong?

Q2: Couldn’t we weaken it *“just a little bit”*?

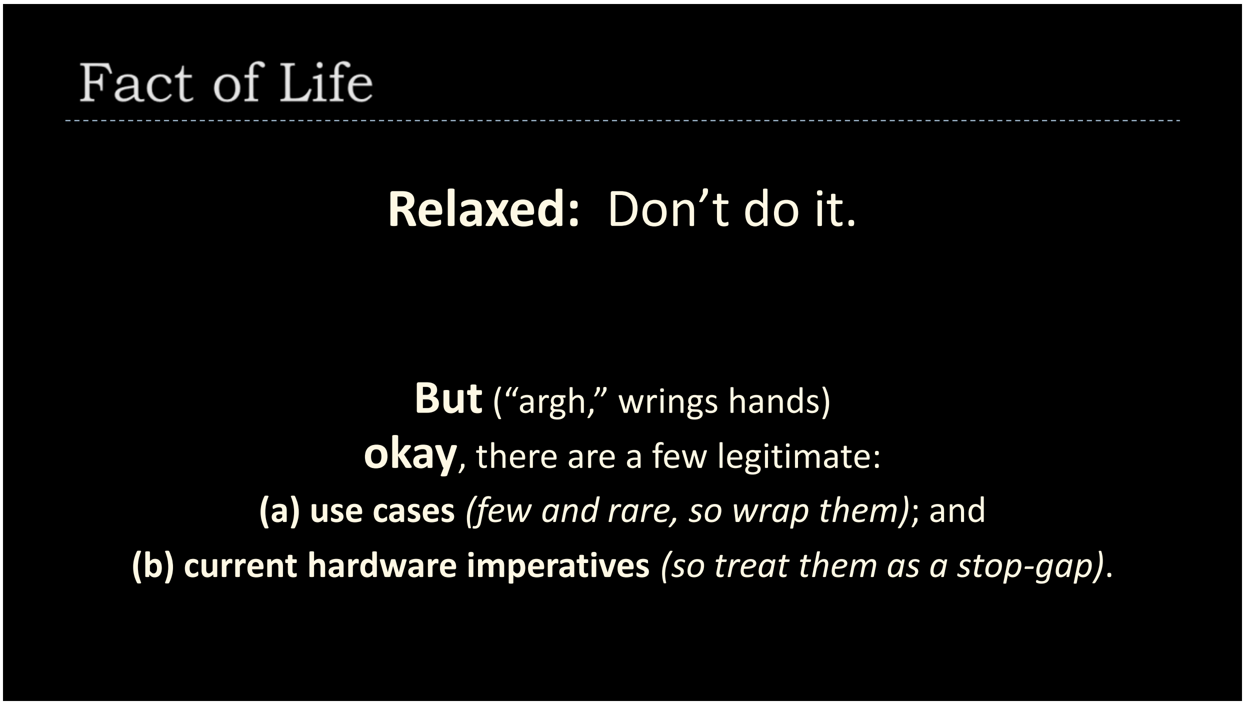
“One Small Step Less Than SC” ?

Fact of Life

Relaxed: Don’t do it.

Data point from Hans Boehm:

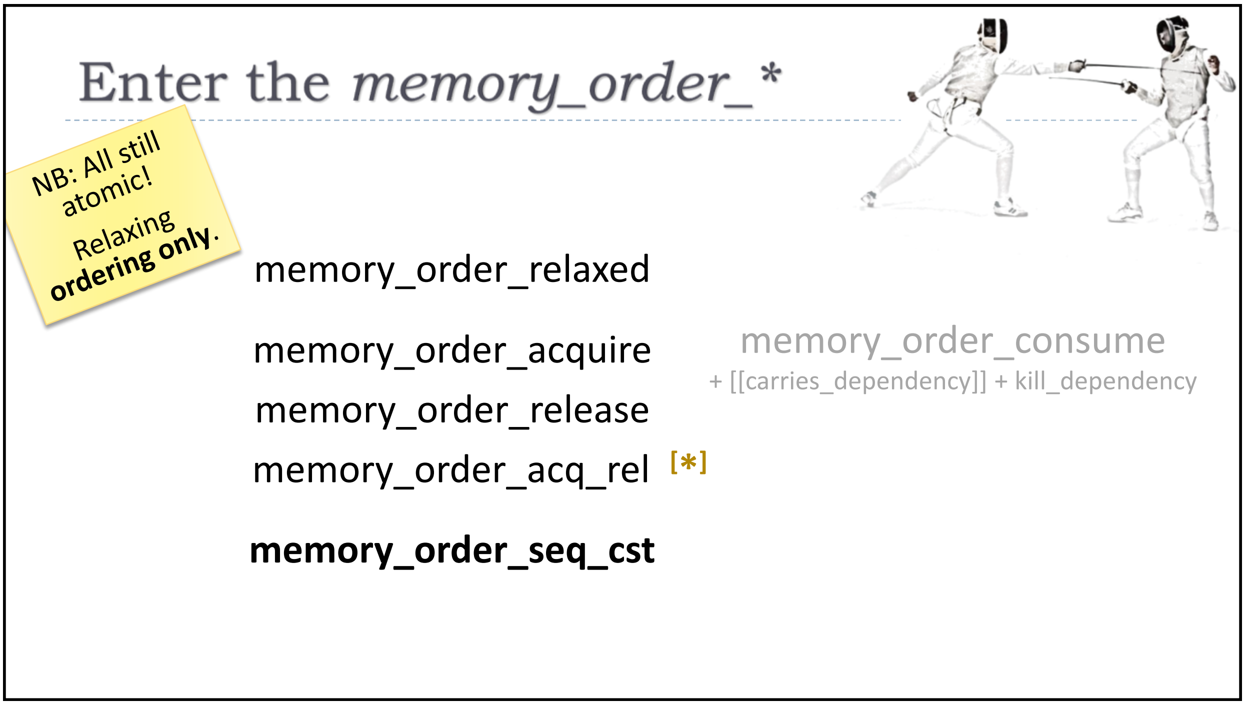
“I would emphasize that we’ve taken great care that **without relaxed atomics, ‘simultaneously’ really means what you thought it did**.”



Fact of Life

Relaxed: Don't do it.

But ("argh," wrings hands)
okay, there are a few legitimate:
(a) use cases *(few and rare, so wrap them)*; and
(b) current hardware imperatives *(so treat them as a stop-gap)*.



Enter the *memory_order_**

NB: All still atomic!
Relaxing **ordering only**.

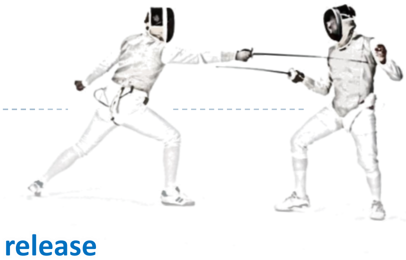
memory_order_relaxed

memory_order_acquire
memory_order_release
memory_order_acq_rel [*]

memory_order_seq_cst

memory_order_consume
+ [[carries_dependency]] + kill_dependency

Enter the *memory_order_**



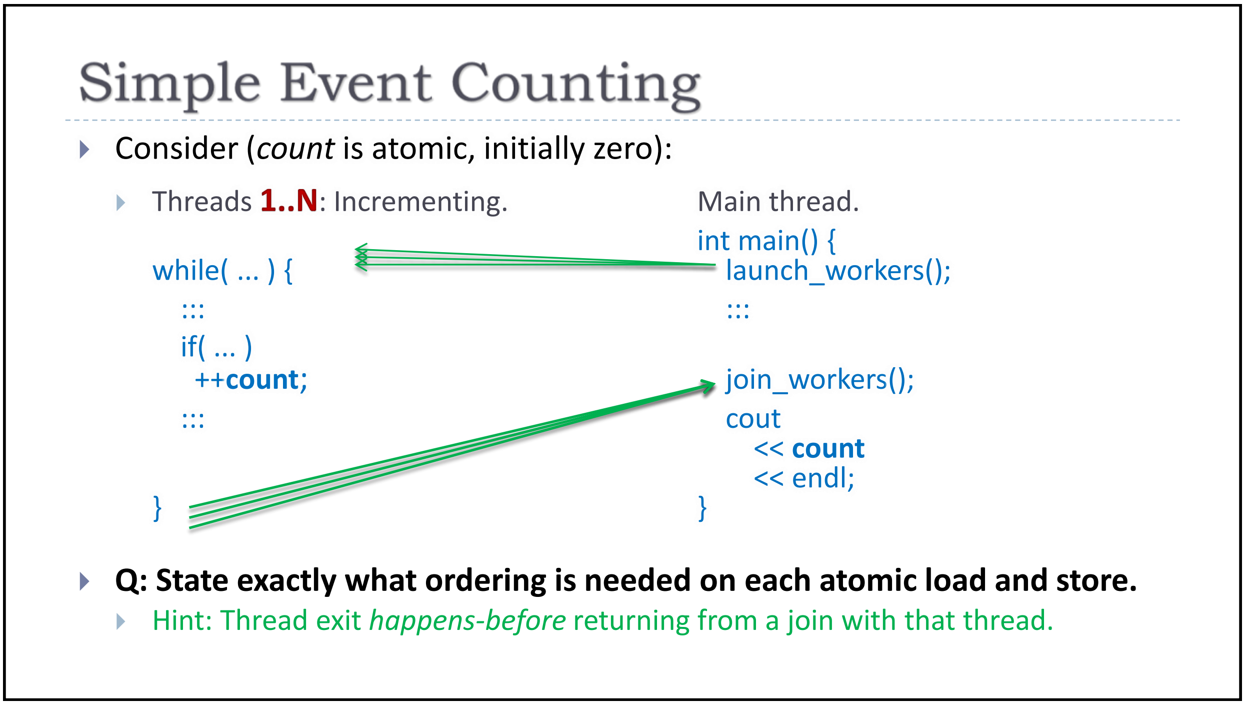
- A word from the Standard (§29.3/1):
  - *memory_order_relaxed*: no operation orders memory
  - *memory_order_release, memory_order_acq_rel, and memory_order_seq_cst*: **a store operation performs a release** operation on the affected memory location
  - *memory_order_consume*: a load operation performs a consume operation on the affected memory location
  - *memory_order_acquire, memory_order_acq_rel, and memory_order_seq_cst*: **a load operation performs an acquire** operation on the affected memory location
- Some combinations are nonsense. Example (§29.6/13):

  C A::load(memory_order order = memory_order_seq_cst) *[…various flavors…]*

  Requires: The order argument shall not be *memory_order_release or memory_order_acq_rel.*

Straying Beyond the Electrified Fence

- A handful of well-known patterns can benefit from judicious use of non-SC atomic operations on some hardware.
  - Examples: Event counters. Dirty flags. Reference counting.
  - Degenerate example: Atomic variable accessed in a race-free manner (i.e., in a region where it doesn't need to be atomic because it's not shared or the program is synchronized in some other way).
- Wrap 'em: Keep the relaxed operations inside types that implement the patterns.
  - "Don't let relaxed atomic op calls spread out into the callers."
  - Problem: It's very subtle to define the library so that the "relaxed-ness" is not detectable to the client.

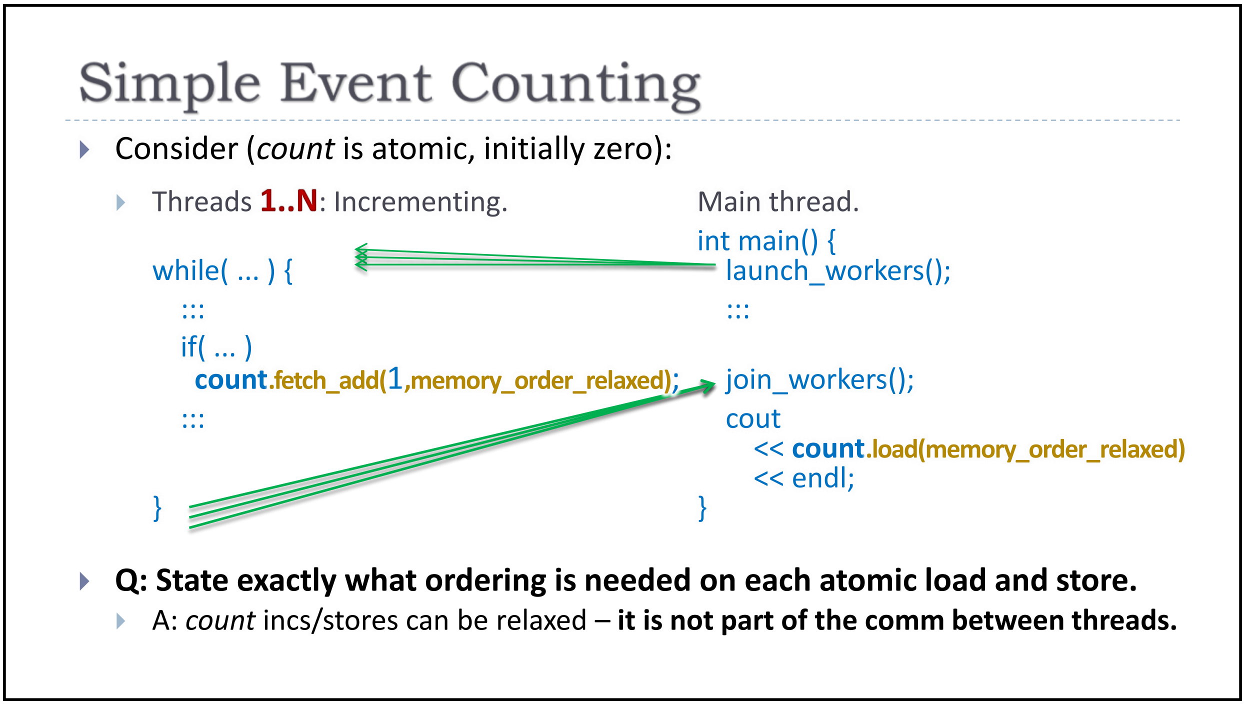


Simple Event Counting

- Consider (*count* is atomic, initially zero):
  - Threads **1..N**: Incrementing. — Main thread.

```
while( ... ) {                    int main() {
  :::                               launch_workers();
  if( ... )                         :::
    ++count;
  :::                               join_workers();
                                    cout
                                      << count
                                      << endl;
}                                 }
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - Hint: Thread exit *happens-before* returning from a join with that thread.

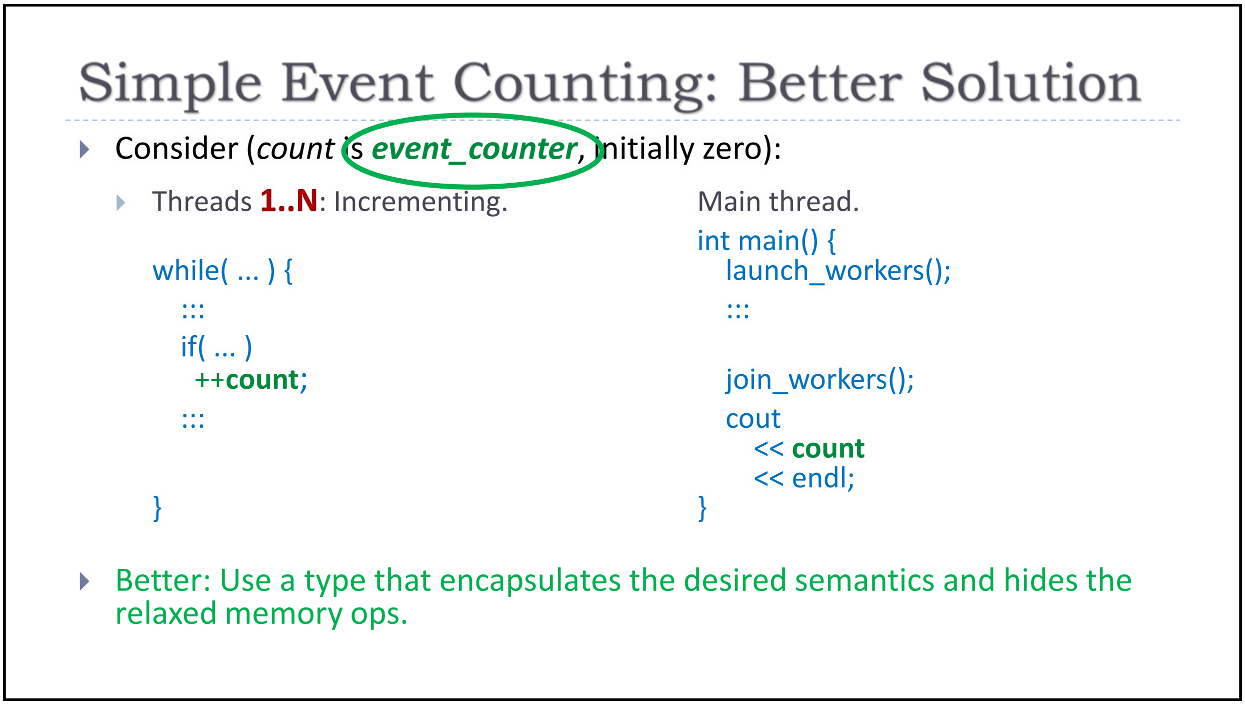


Simple Event Counting

- Consider (*count* is atomic, initially zero):
  - Threads **1..N**: Incrementing. — Main thread.

```
while( ... ) {                    int main() {
  :::                               launch_workers();
  if( ... )                         :::
    count.fetch_add(1,memory_order_relaxed);
                                    join_workers();
  :::                               cout
                                      << count.load(memory_order_relaxed)
                                      << endl;
}                                 }
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - A: *count* incs/stores can be relaxed – **it is not part of the comm between threads.**



Simple Event Counting: Better Solution

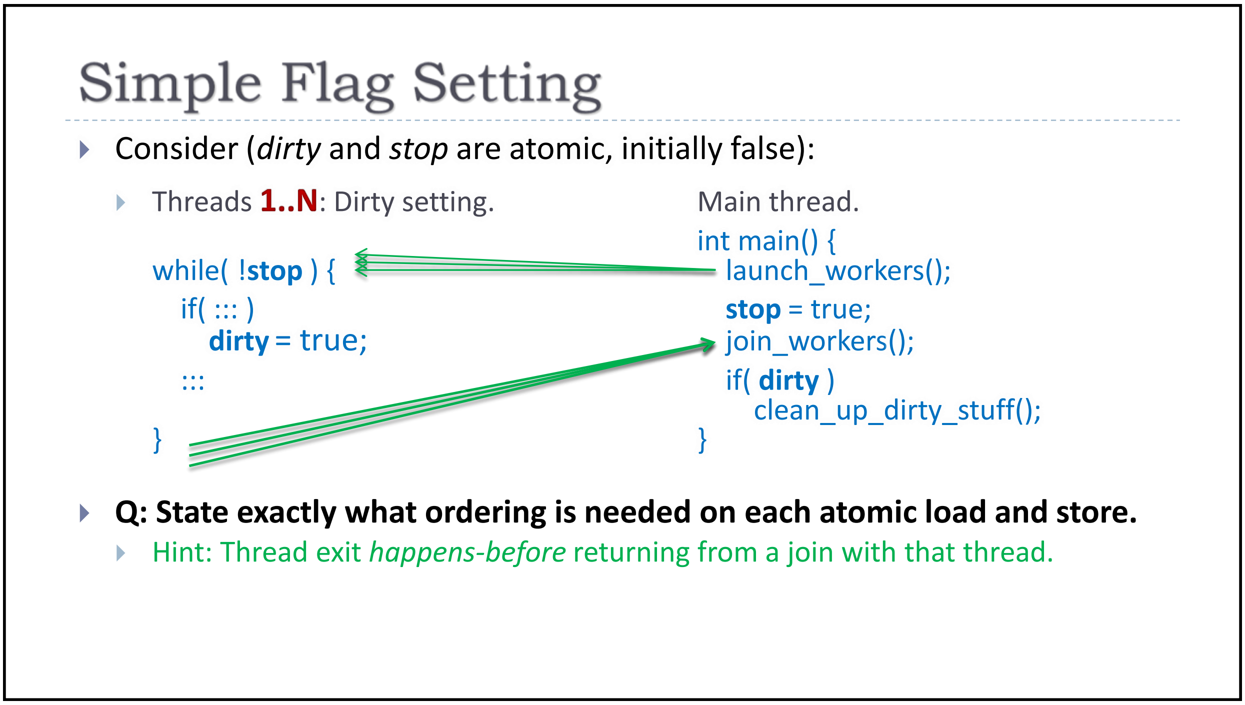
- Consider (*count* is ***event_counter***, initially zero):
  - Threads **1..N**: Incrementing.

```
while( ... ) {
   :::
   if( ... )
     ++count;
   :::


}
```

Main thread.

```
int main() {
   launch_workers();
   :::


   join_workers();
   cout
      << count
      << endl;
}
```

- Better: Use a type that encapsulates the desired semantics and hides the relaxed memory ops.



Simple Flag Setting

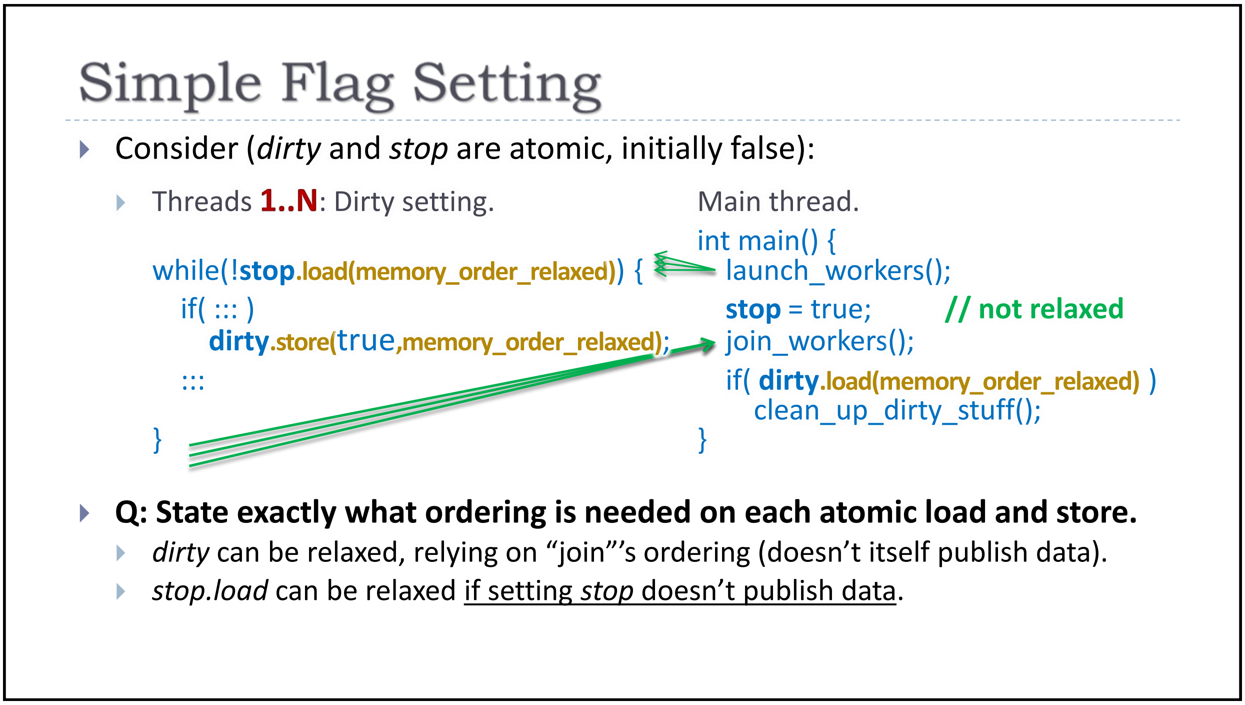
- Consider (*dirty* and *stop* are atomic, initially false):
  - Threads **1..N**: Dirty setting.

```
while( !stop ) {
   if( ::: )
      dirty = true;
   :::

}
```

Main thread.

```
int main() {
   launch_workers();
   stop = true;
   join_workers();
   if( dirty )
      clean_up_dirty_stuff();
}
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - Hint: Thread exit *happens-before* returning from a join with that thread.



Simple Flag Setting

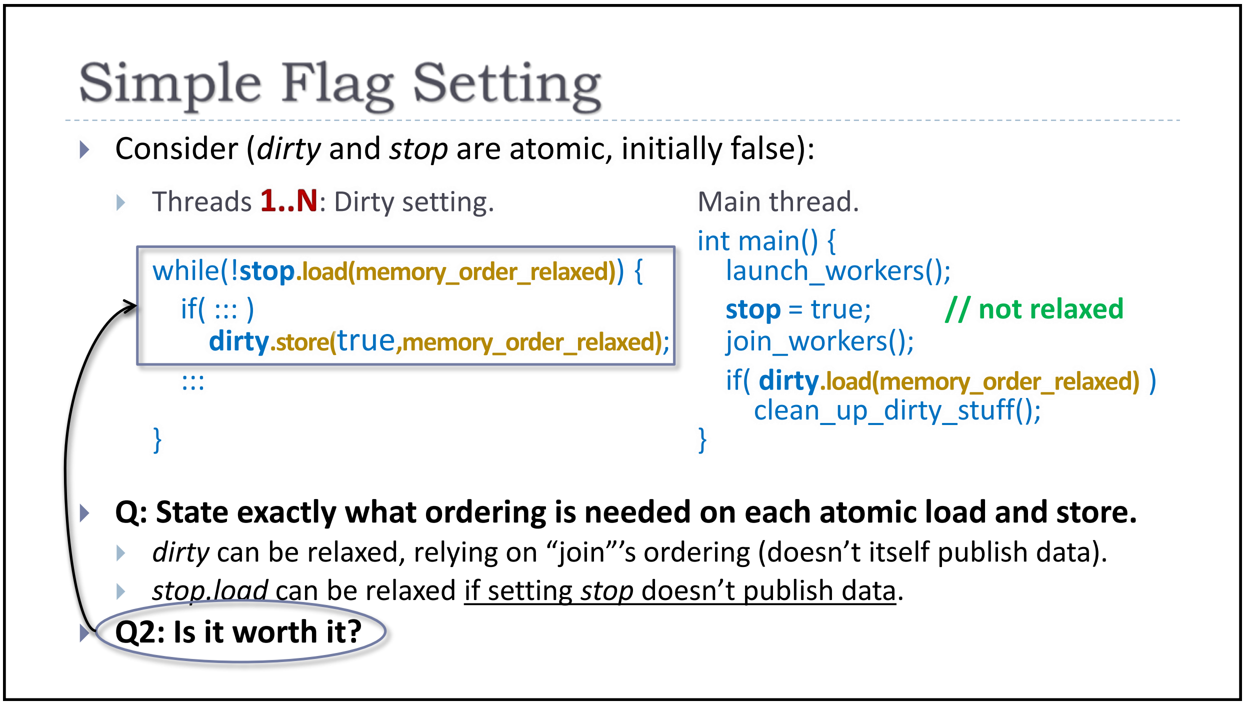
- Consider (*dirty* and *stop* are atomic, initially false):
  - Threads **1..N**: Dirty setting.

```
while(!stop.load(memory_order_relaxed)) {
  if( ::: )
    dirty.store(true,memory_order_relaxed);
  :::
}
```

Main thread.

```
int main() {
  launch_workers();
  stop = true;          // not relaxed
  join_workers();
  if( dirty.load(memory_order_relaxed) )
    clean_up_dirty_stuff();
}
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - *dirty* can be relaxed, relying on "join"'s ordering (doesn't itself publish data).
  - *stop.load* can be relaxed if setting *stop* doesn't publish data.



Simple Flag Setting

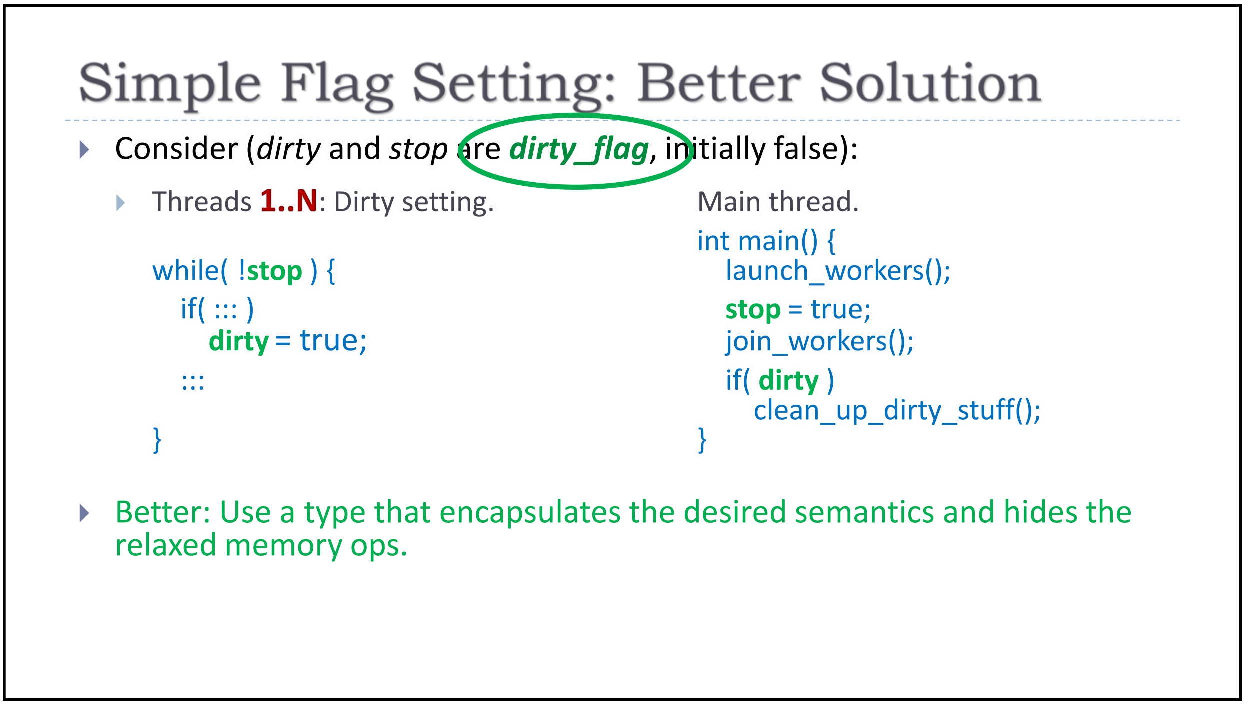
- Consider (*dirty* and *stop* are atomic, initially false):
  - Threads **1..N**: Dirty setting.

```
while(!stop.load(memory_order_relaxed)) {
  if( ::: )
    dirty.store(true,memory_order_relaxed);
  :::
}
```

Main thread.

```
int main() {
  launch_workers();
  stop = true;          // not relaxed
  join_workers();
  if( dirty.load(memory_order_relaxed) )
    clean_up_dirty_stuff();
}
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - *dirty* can be relaxed, relying on "join"'s ordering (doesn't itself publish data).
  - *stop.load* can be relaxed if setting *stop* doesn't publish data.
- **Q2: Is it worth it?**



Simple Flag Setting: Better Solution

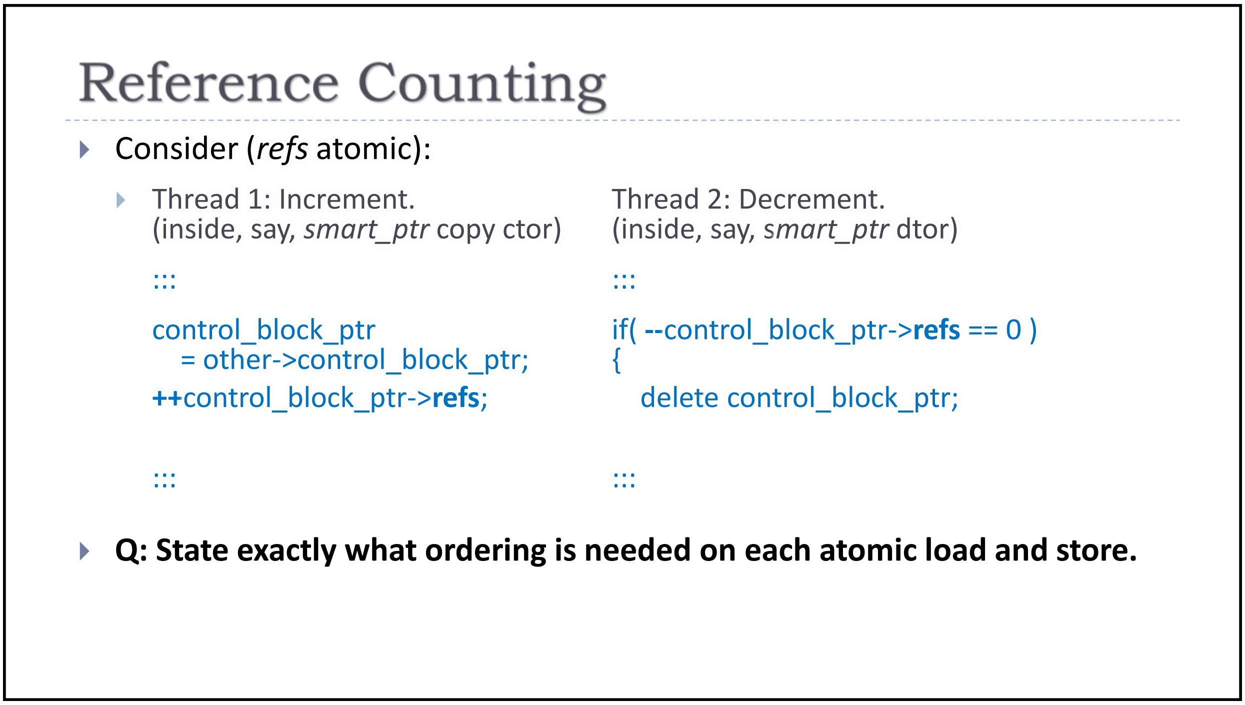
- Consider (*dirty* and *stop* are ***dirty_flag***, initially false):
  - Threads **1..N**: Dirty setting.

```
while( !stop ) {
   if( ::: )
      dirty = true;
   :::
}
```

Main thread.

```
int main() {
   launch_workers();
   stop = true;
   join_workers();
   if( dirty )
      clean_up_dirty_stuff();
}
```

- Better: Use a type that encapsulates the desired semantics and hides the relaxed memory ops.



Reference Counting

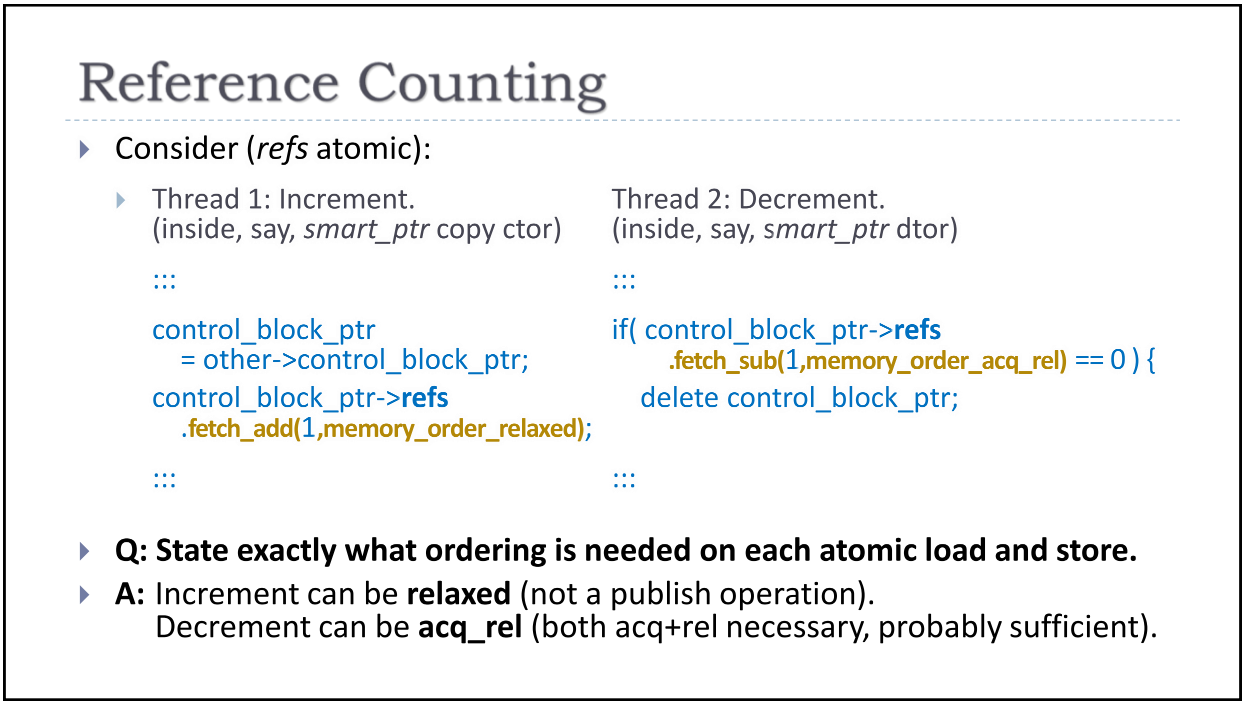
- Consider (*refs* atomic):
  - Thread 1: Increment. (inside, say, *smart_ptr* copy ctor)

```
:::
control_block_ptr
   = other->control_block_ptr;
++control_block_ptr->refs;

:::
```

Thread 2: Decrement. (inside, say, *smart_ptr* dtor)

```
:::
if( --control_block_ptr->refs == 0 )
{
   delete control_block_ptr;

:::
```

- **Q: State exactly what ordering is needed on each atomic load and store.**



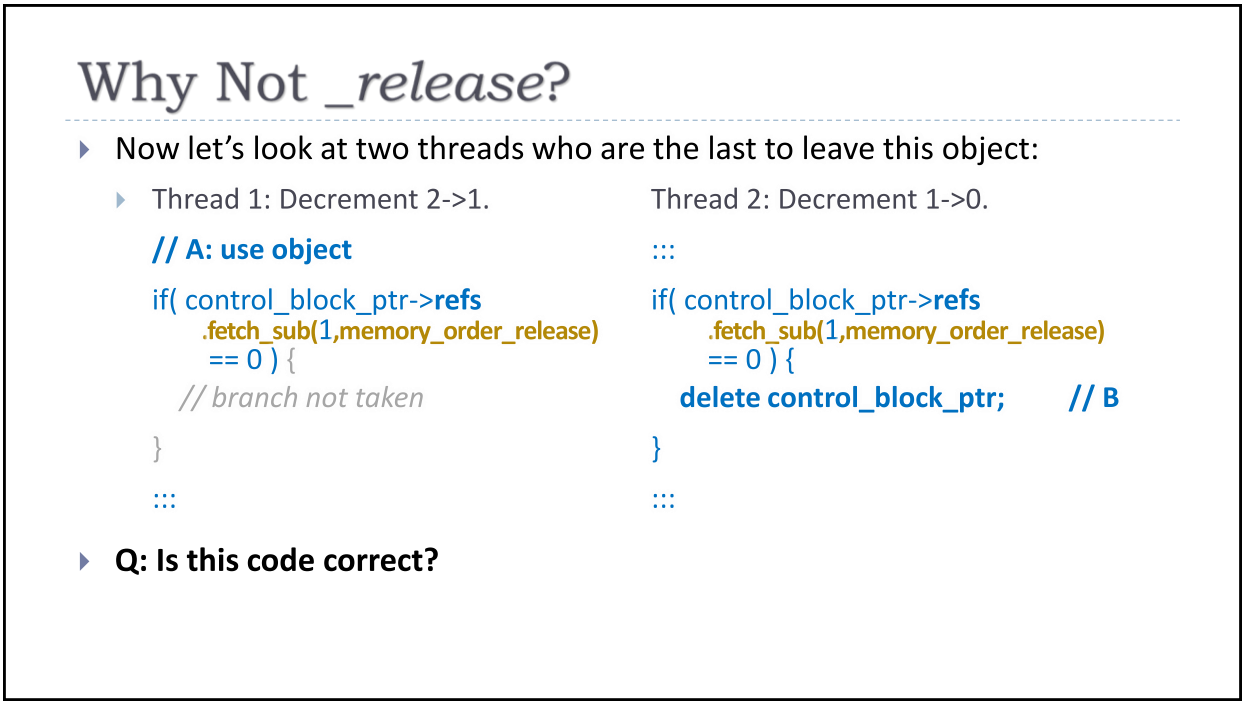
Reference Counting

- Consider (*refs* atomic):
  - Thread 1: Increment.
    (inside, say, *smart_ptr* copy ctor)

    ```
    :::
    control_block_ptr
       = other->control_block_ptr;
    control_block_ptr->refs
       .fetch_add(1,memory_order_relaxed);
    :::
    ```

    Thread 2: Decrement.
    (inside, say, *smart_ptr* dtor)

    ```
    :::
    if( control_block_ptr->refs
          .fetch_sub(1,memory_order_acq_rel) == 0 ) {
       delete control_block_ptr;
    :::
    ```
- **Q: State exactly what ordering is needed on each atomic load and store.**
- **A:** Increment can be **relaxed** (not a publish operation).
  Decrement can be **acq_rel** (both acq+rel necessary, probably sufficient).



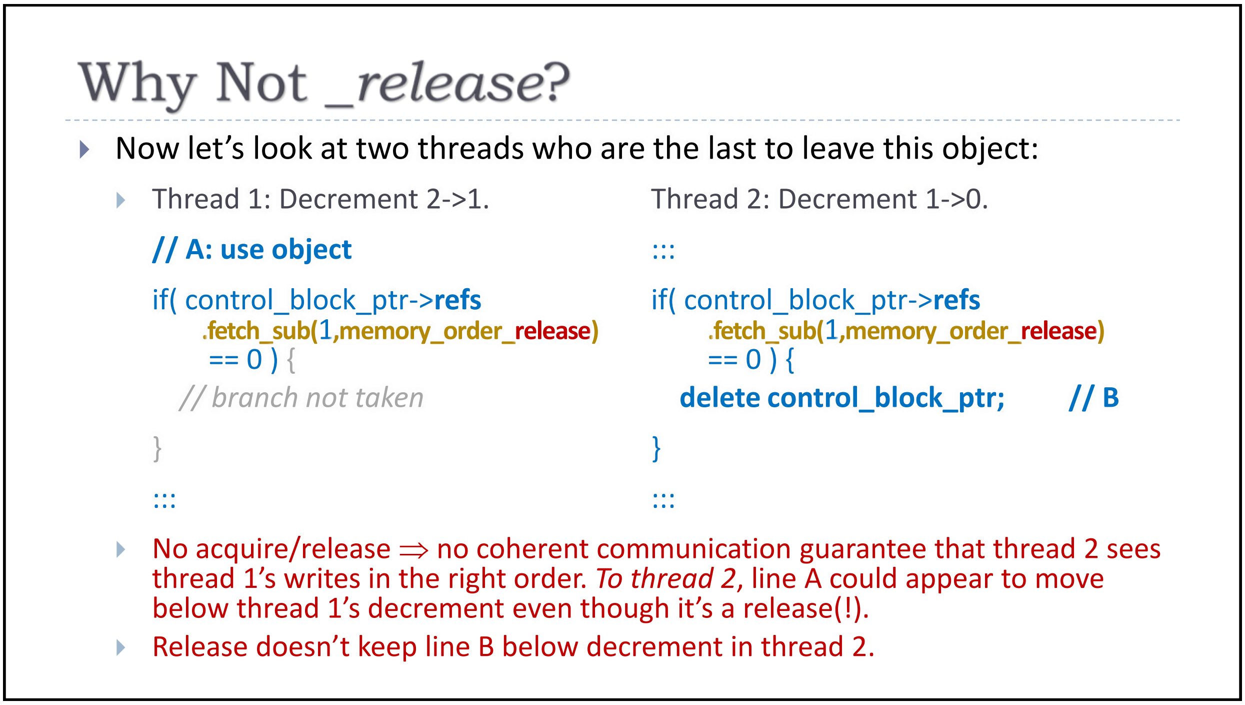
Why Not *_release*?

- Now let's look at two threads who are the last to leave this object:
  - Thread 1: Decrement 2->1.

    ```
    // A: use object
    if( control_block_ptr->refs
          .fetch_sub(1,memory_order_release)
          == 0 ) {
       // branch not taken
    }
    :::
    ```

    Thread 2: Decrement 1->0.

    ```
    :::
    if( control_block_ptr->refs
          .fetch_sub(1,memory_order_release)
          == 0 ) {
       delete control_block_ptr;       // B
    }
    :::
    ```
- **Q: Is this code correct?**



Why Not _release?

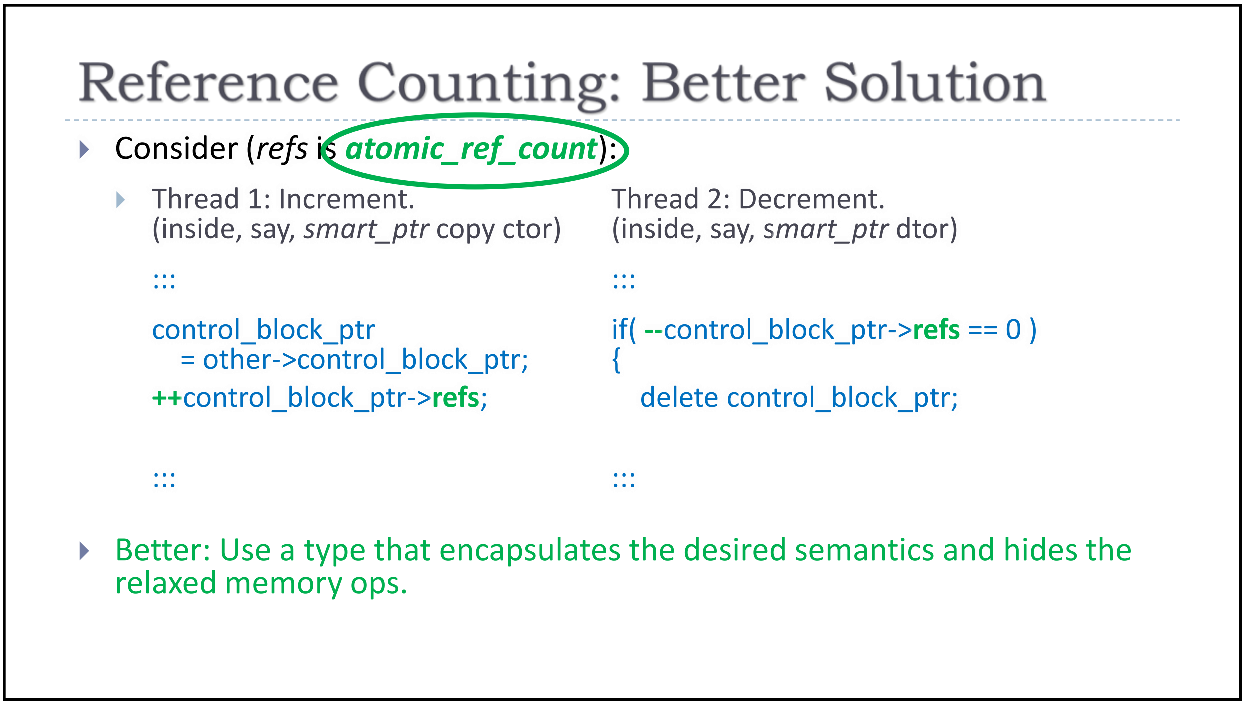
- Now let's look at two threads who are the last to leave this object:
  - Thread 1: Decrement 2->1.

```
// A: use object

if( control_block_ptr->refs
      .fetch_sub(1,memory_order_release)
      == 0 ) {
   // branch not taken

}

:::
```

Thread 2: Decrement 1->0.

```
:::

if( control_block_ptr->refs
      .fetch_sub(1,memory_order_release)
      == 0 ) {
   delete control_block_ptr;        // B

}

:::
```

  - No acquire/release ⇒ no coherent communication guarantee that thread 2 sees thread 1's writes in the right order. *To thread 2*, line A could appear to move below thread 1's decrement even though it's a release(!).
  - Release doesn't keep line B below decrement in thread 2.



Reference Counting: Better Solution

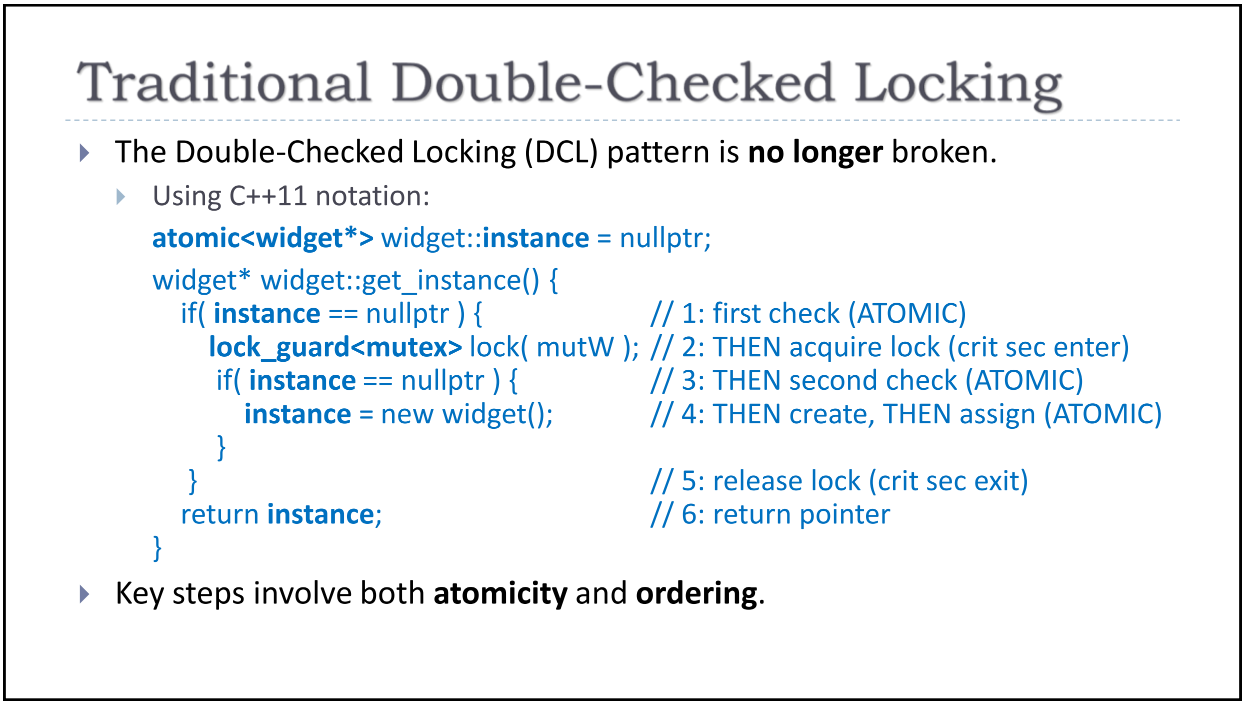
- Consider (*refs* is ***atomic_ref_count***):
  - Thread 1: Increment.
    (inside, say, *smart_ptr* copy ctor)

```
:::

control_block_ptr
   = other->control_block_ptr;
++control_block_ptr->refs;


:::
```

Thread 2: Decrement.
(inside, say, *smart_ptr* dtor)

```
:::

if( --control_block_ptr->refs == 0 )
{
   delete control_block_ptr;


:::
```

- Better: Use a type that encapsulates the desired semantics and hides the relaxed memory ops.

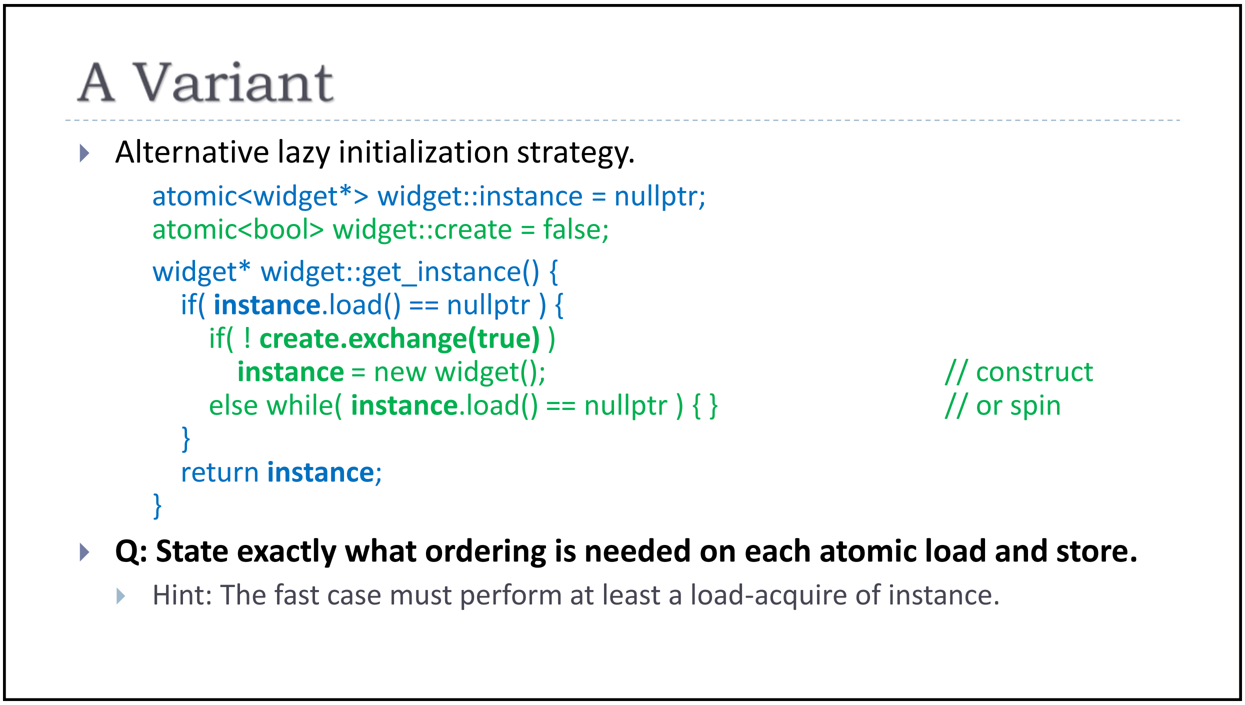


Traditional Double-Checked Locking

- The Double-Checked Locking (DCL) pattern is **no longer** broken.
  - Using C++11 notation:

```
atomic<widget*> widget::instance = nullptr;

widget* widget::get_instance() {
  if( instance == nullptr ) {               // 1: first check (ATOMIC)
    lock_guard<mutex> lock( mutW );  // 2: THEN acquire lock (crit sec enter)
     if( instance == nullptr ) {            // 3: THEN second check (ATOMIC)
       instance = new widget();             // 4: THEN create, THEN assign (ATOMIC)
     }
   }                                        // 5: release lock (crit sec exit)
  return instance;                          // 6: return pointer
}
```

- Key steps involve both **atomicity** and **ordering**.

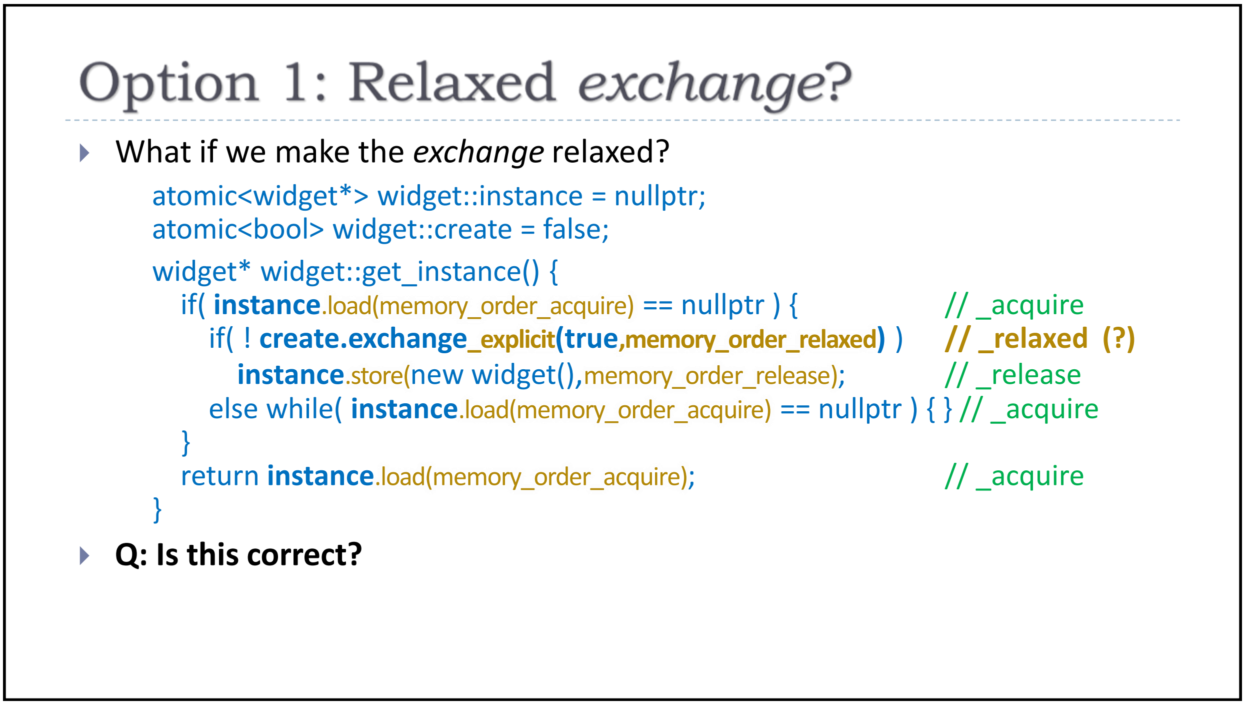


A Variant

- Alternative lazy initialization strategy.

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance.load() == nullptr ) {
    if( ! create.exchange(true) )
      instance = new widget();                          // construct
    else while( instance.load() == nullptr ) { }        // or spin
  }
  return instance;
}
```

- **Q: State exactly what ordering is needed on each atomic load and store.**
  - Hint: The fast case must perform at least a load-acquire of instance.

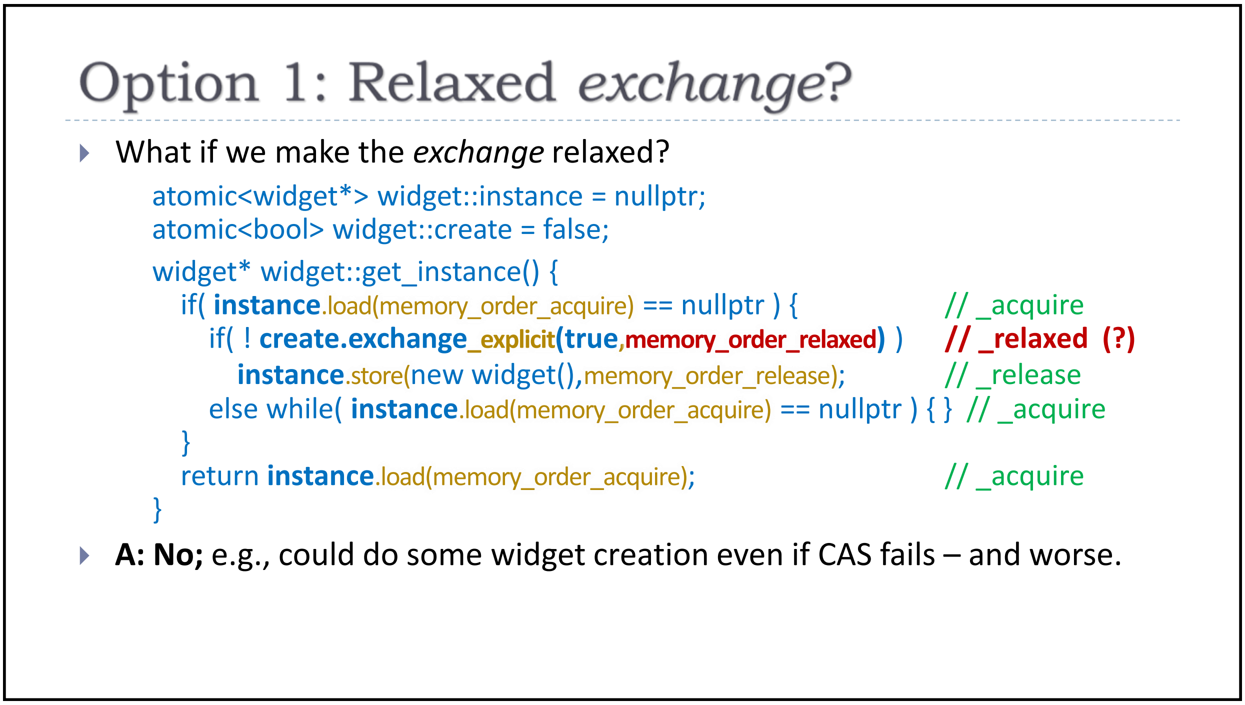


Option 1: Relaxed *exchange*?

- What if we make the *exchange* relaxed?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance.load(memory_order_acquire) == nullptr ) {          // _acquire
    if( ! create.exchange_explicit(true,memory_order_relaxed) )   // _relaxed  (?)
      instance.store(new widget(),memory_order_release);          // _release
    else while( instance.load(memory_order_acquire) == nullptr ) { } // _acquire
  }
  return instance.load(memory_order_acquire);                     // _acquire
}
```

- **Q: Is this correct?**

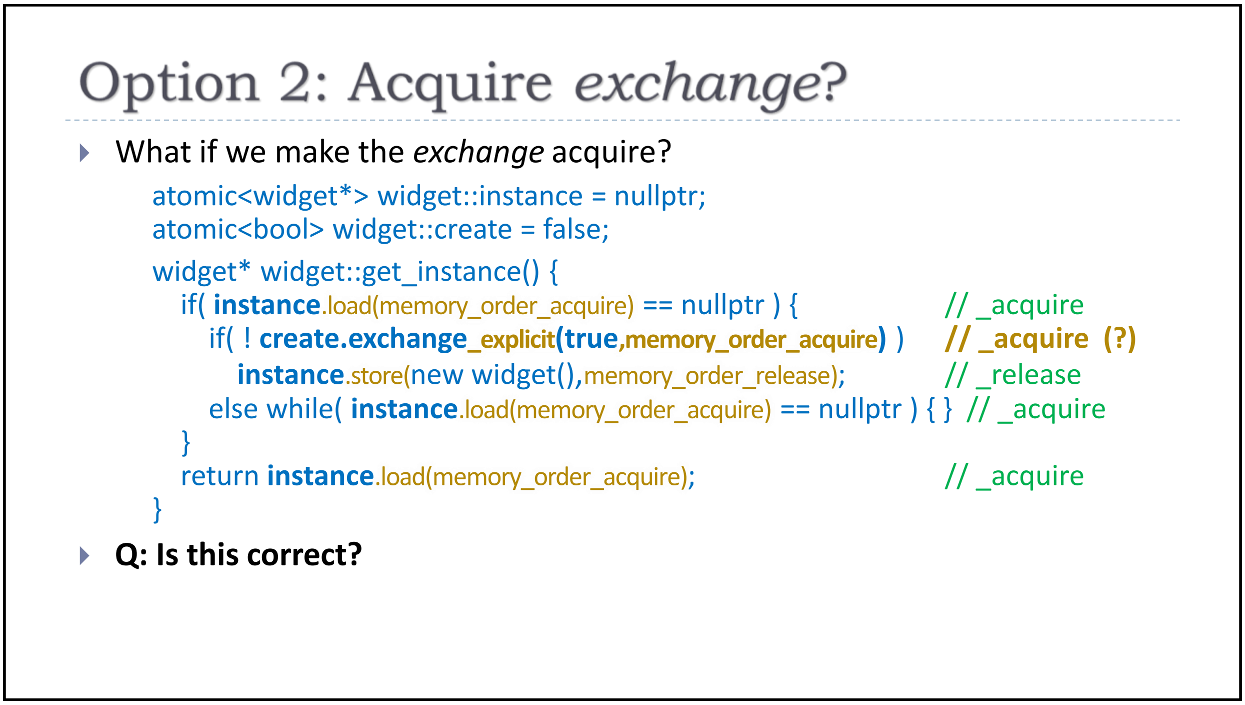


Option 1: Relaxed *exchange*?

- What if we make the *exchange* relaxed?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance.load(memory_order_acquire) == nullptr ) {          // _acquire
    if( ! create.exchange_explicit(true,memory_order_relaxed) )   // _relaxed  (?)
      instance.store(new widget(),memory_order_release);          // _release
    else while( instance.load(memory_order_acquire) == nullptr ) { }  // _acquire
  }
  return instance.load(memory_order_acquire);                     // _acquire
}
```

- **A: No;** e.g., could do some widget creation even if CAS fails – and worse.

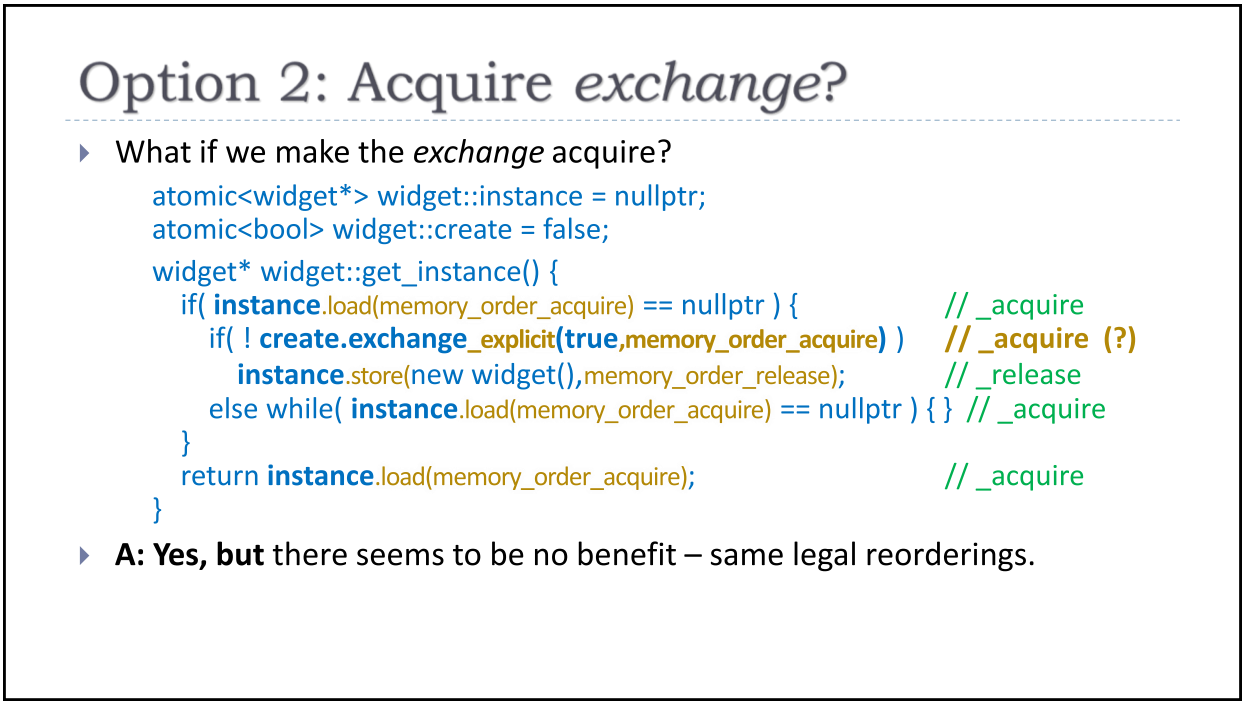


Option 2: Acquire *exchange*?

- What if we make the *exchange* acquire?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance.load(memory_order_acquire) == nullptr ) {            // _acquire
    if( ! create.exchange_explicit(true,memory_order_acquire) )    // _acquire  (?)
      instance.store(new widget(),memory_order_release);           // _release
    else while( instance.load(memory_order_acquire) == nullptr ) { }  // _acquire
  }
  return instance.load(memory_order_acquire);                      // _acquire
}
```

- **Q: Is this correct?**

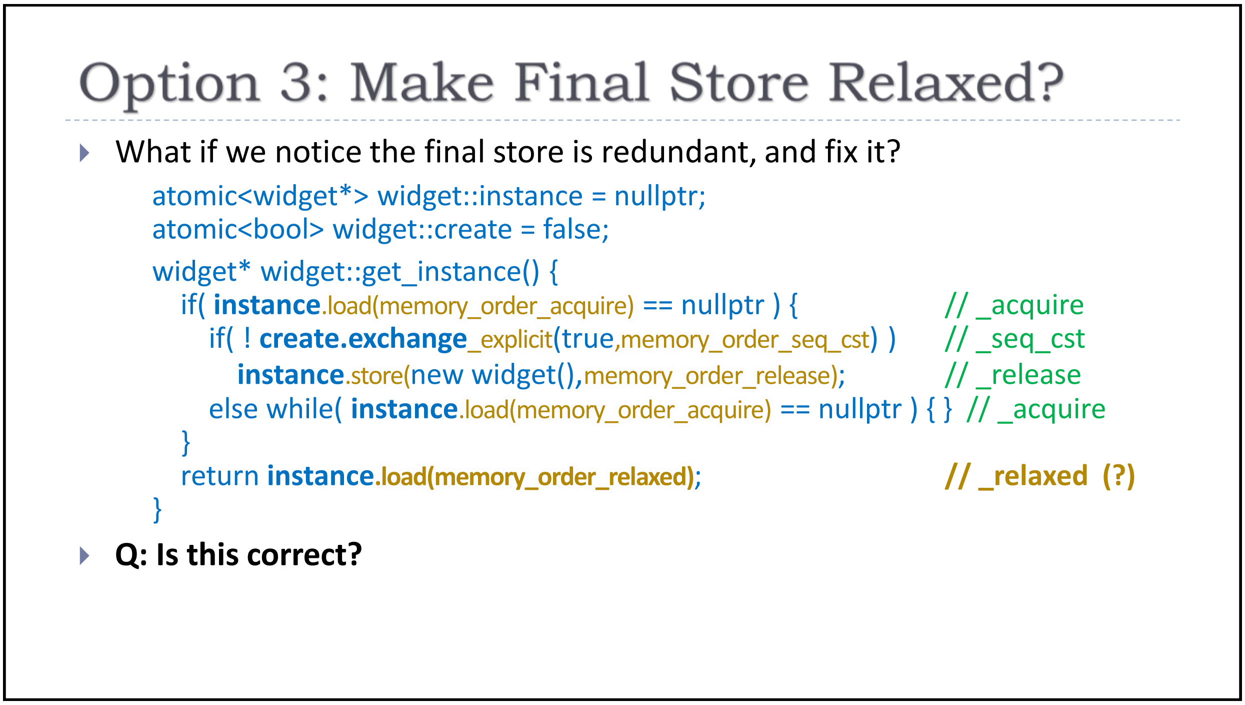


Option 2: Acquire *exchange*?

- What if we make the *exchange* acquire?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance.load(memory_order_acquire) == nullptr ) {            // _acquire
    if( ! create.exchange_explicit(true,memory_order_acquire) )    // _acquire  (?)
      instance.store(new widget(),memory_order_release);           // _release
    else while( instance.load(memory_order_acquire) == nullptr ) { }  // _acquire
  }
  return instance.load(memory_order_acquire);                      // _acquire
}
```

- **A: Yes, but** there seems to be no benefit – same legal reorderings.

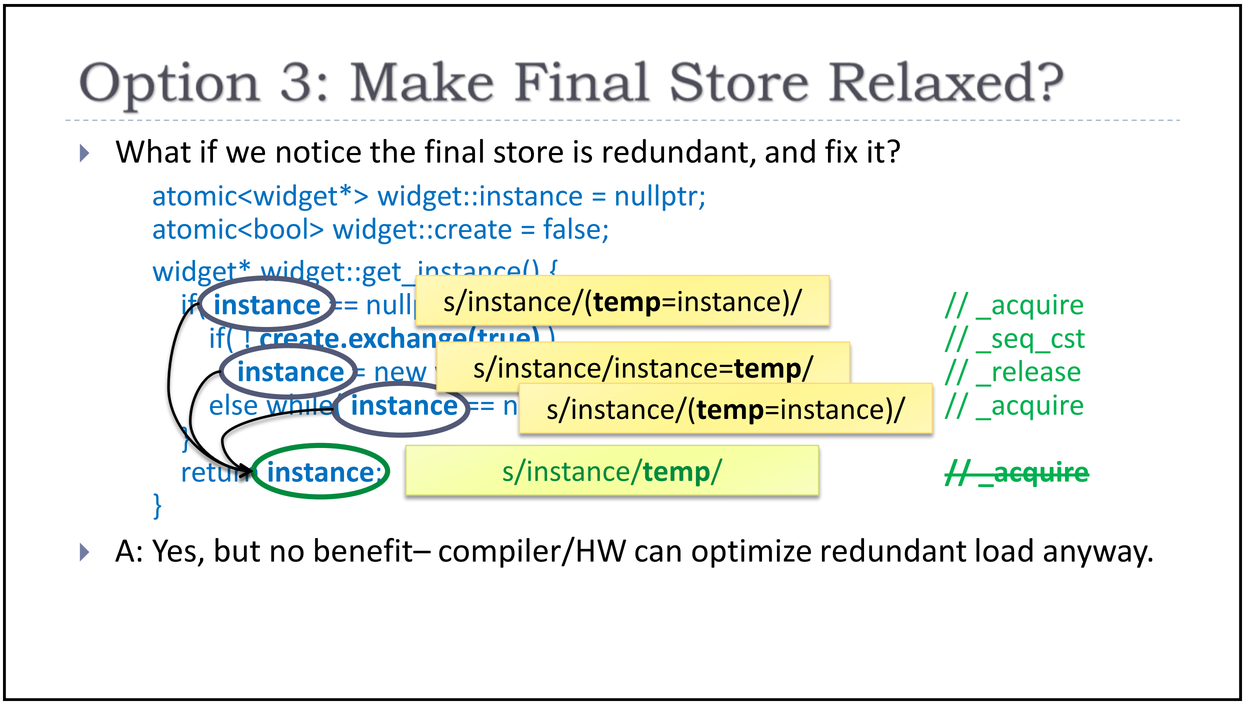


Option 3: Make Final Store Relaxed?

- What if we notice the final store is redundant, and fix it?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance.load(memory_order_acquire) == nullptr ) {          // _acquire
    if( ! create.exchange_explicit(true,memory_order_seq_cst) )   // _seq_cst
      instance.store(new widget(),memory_order_release);          // _release
    else while( instance.load(memory_order_acquire) == nullptr ) { }  // _acquire
  }
  return instance.load(memory_order_relaxed);                     // _relaxed  (?)
}
```

- **Q: Is this correct?**



Option 3: Make Final Store Relaxed?

- What if we notice the final store is redundant, and fix it?

```
atomic<widget*> widget::instance = nullptr;
atomic<bool> widget::create = false;

widget* widget::get_instance() {
  if( instance == null                        // _acquire
    if( ! create.exchange(true) )             // _seq_cst
      instance = new                          // _release
    else while( instance == n                 // _acquire
  }
  return instance;                            ~~// _acquire~~
}
```

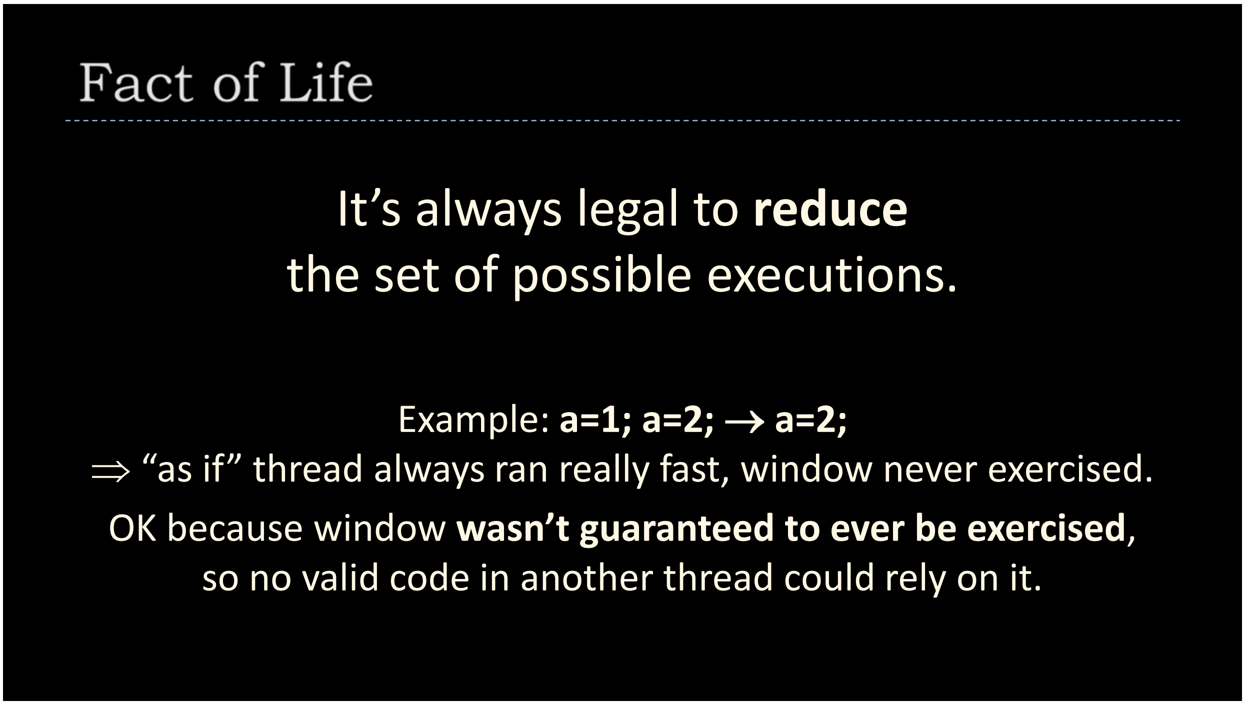
s/instance/(**temp**=instance)/

s/instance/instance=**temp**/

s/instance/(**temp**=instance)/

s/instance/**temp**/

- A: Yes, but no benefit– compiler/HW can optimize redundant load anyway.



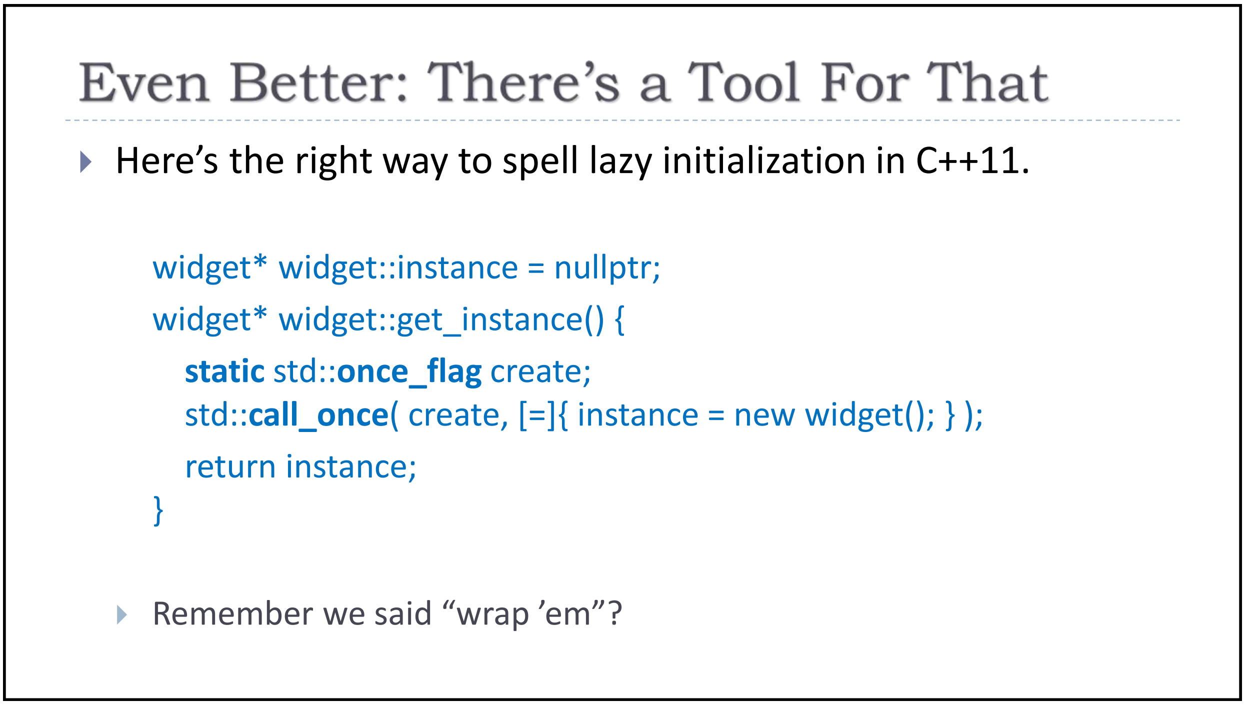
Fact of Life

It’s always legal to **reduce** the set of possible executions.

Example: **a=1; a=2; → a=2;**

⇒ “as if” thread always ran really fast, window never exercised.

OK because window **wasn’t guaranteed to ever be exercised**, so no valid code in another thread could rely on it.

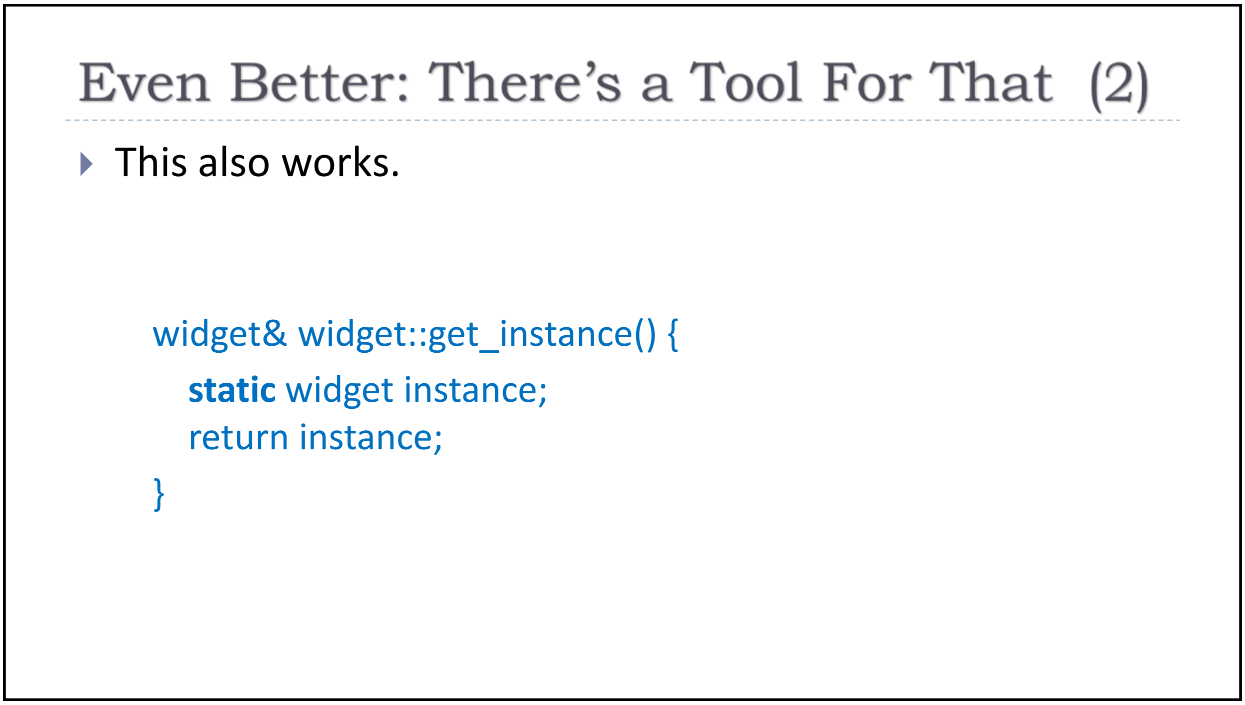


Even Better: There’s a Tool For That

- Here’s the right way to spell lazy initialization in C++11.

```
widget* widget::instance = nullptr;
widget* widget::get_instance() {
   static std::once_flag create;
   std::call_once( create, [=]{ instance = new widget(); } );
   return instance;
}
```

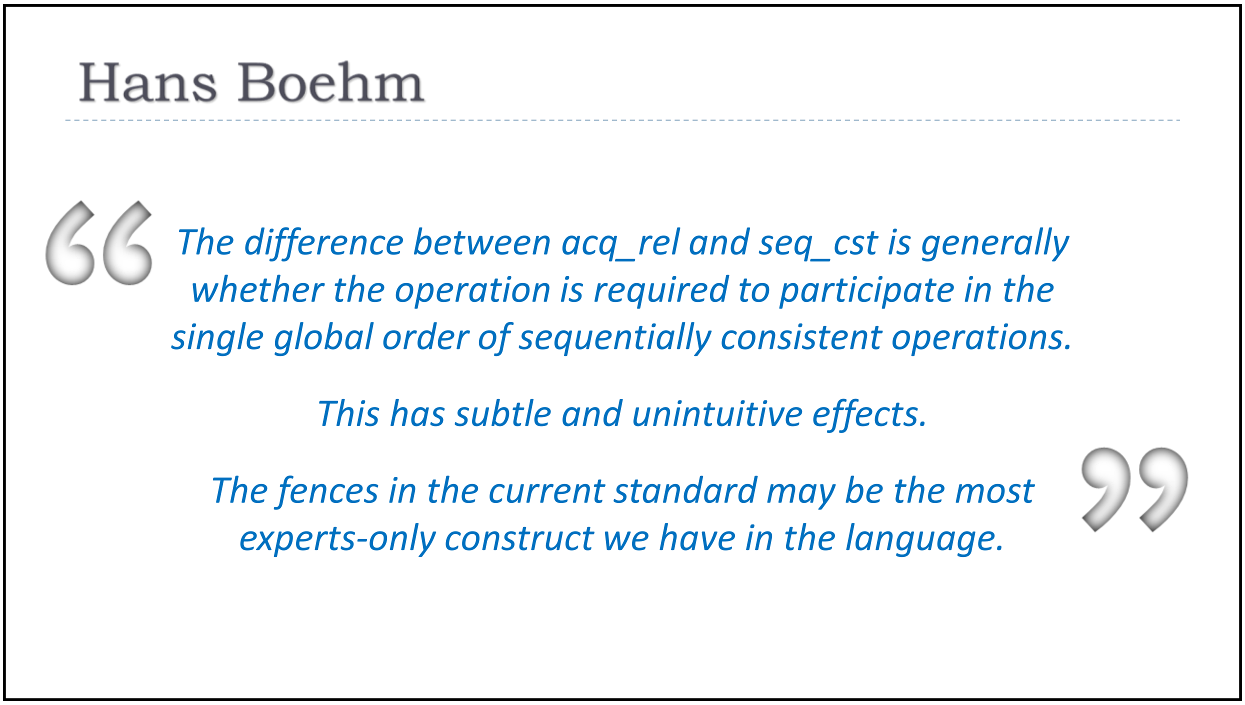
- Remember we said “wrap ’em”?



Even Better: There's a Tool For That (2)

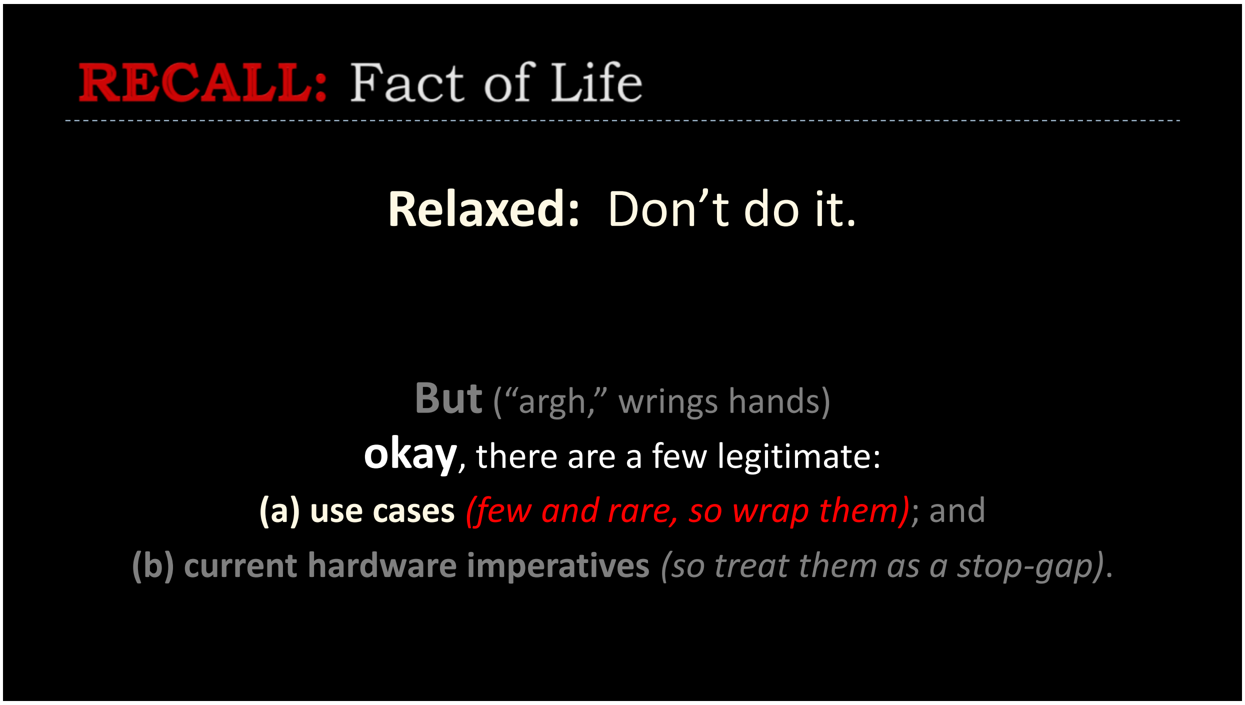
- This also works.

```
widget& widget::get_instance() {
    static widget instance;
    return instance;
}
```



Hans Boehm

> *The difference between acq_rel and seq_cst is generally whether the operation is required to participate in the single global order of sequentially consistent operations.*
>
> *This has subtle and unintuitive effects.*
>
> *The fences in the current standard may be the most experts-only construct we have in the language.*



RECALL: Fact of Life

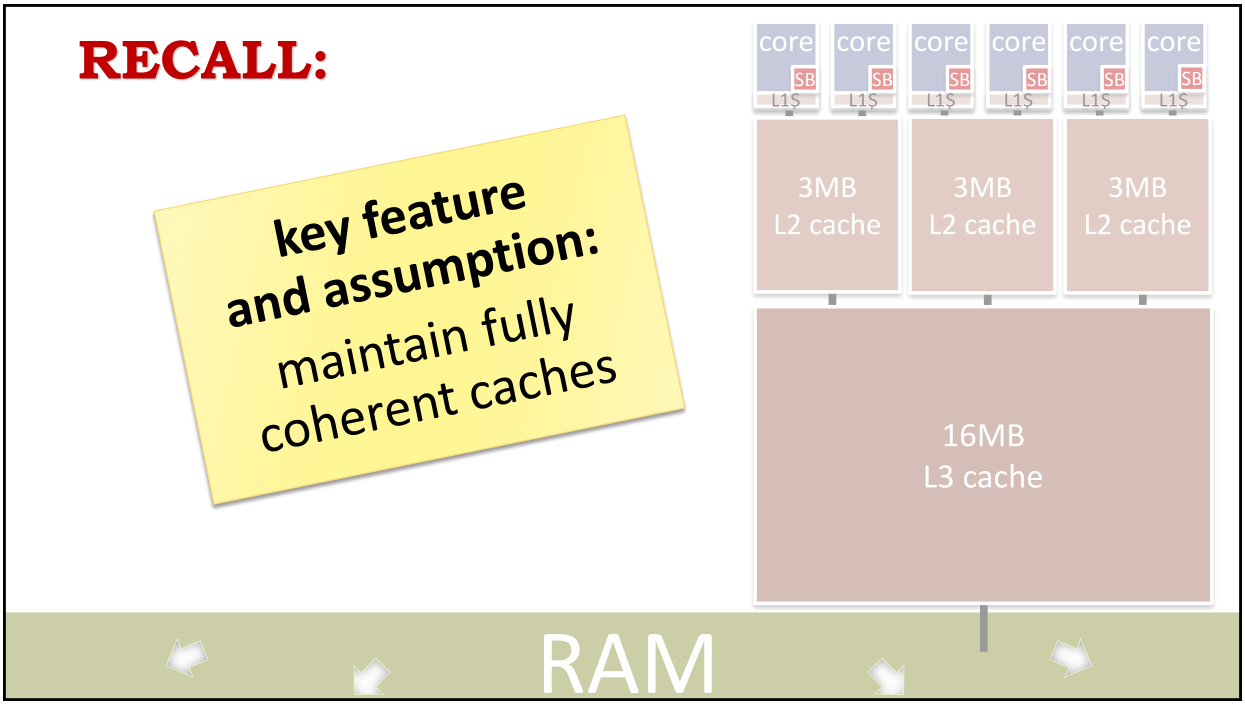
Relaxed: Don't do it.

But ("argh," wrings hands)

okay, there are a few legitimate:

(a) use cases *(few and rare, so wrap them)*; and

(b) current hardware imperatives *(so treat them as a stop-gap)*.



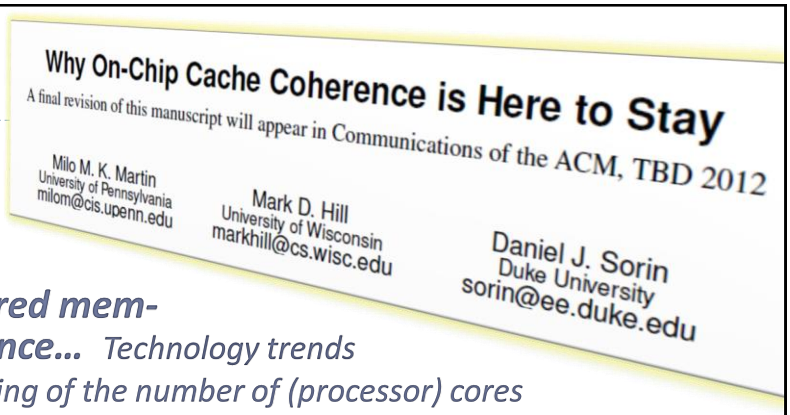
RECALL:

key feature and assumption:

maintain fully coherent caches

Good News!

- **July 2012 *CACM*:**



Why On-Chip Cache Coherence is Here to Stay

A final revision of this manuscript will appear in Communications of the ACM, TBD 2012

Milo M. K. Martin
University of Pennsylvania
milom@cis.upenn.edu

Mark D. Hill
University of Wisconsin
markhill@cs.wisc.edu

Daniel J. Sorin
Duke University
sorin@ee.duke.edu

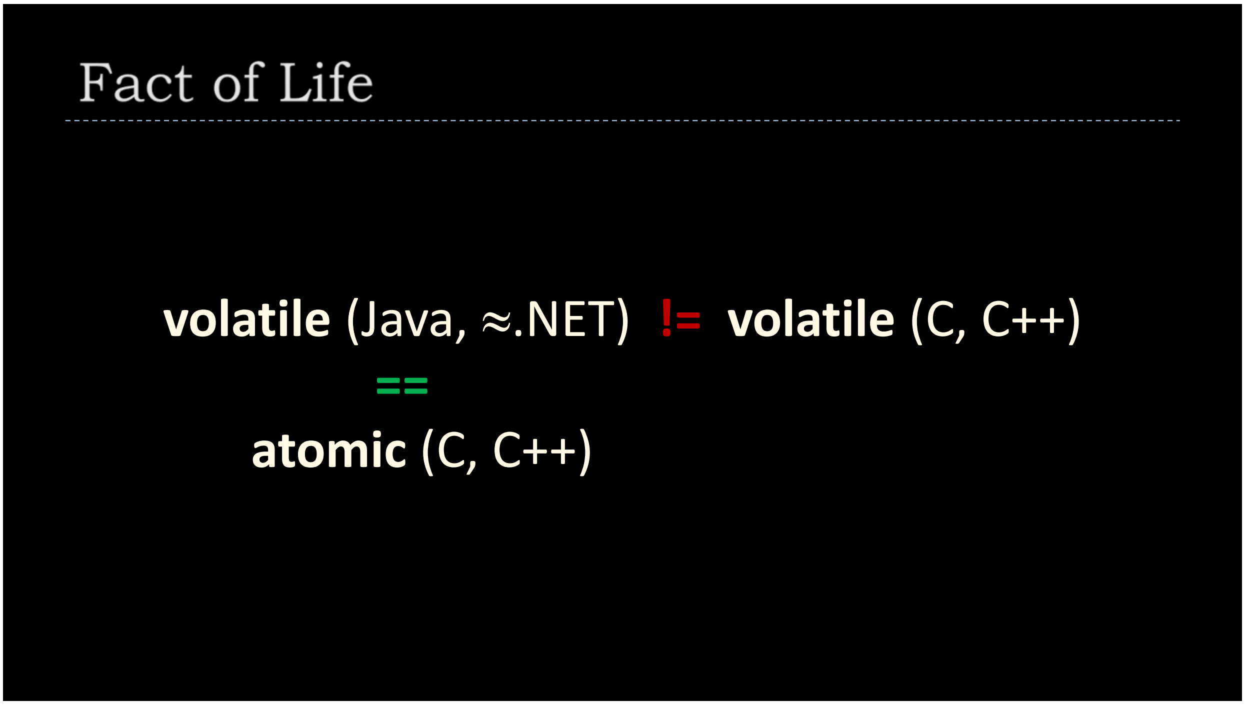
*Today's multicore chips commonly implement **shared memory with cache coherence…** Technology trends continue to enable the scaling of the number of (processor) cores per chip. Because conventional wisdom says that the coherence does not scale well to many cores, some prognosticators predict the end of coherence.*

*This paper refutes this conventional wisdom… we predict that **on-chip coherence and the programming convenience and compatibility it provides are here to stay.***

http://www.cis.upenn.edu/acg/papers/cacm12_why_coherence_nearfinal.pdf

Roadmap

- Optimizations, Races, and the Memory Model
- Ordering – What: Acquire and Release
- Ordering – How: Mutexes, Atomics, and/or Fences
- Other Restrictions on Compilers and Hardware (→ Bugs)
- Code Gen & Performance: x86/x64, IA64, POWER, ARM, ... ???
- Relaxed Atomics (as time allows)
- Coda: Volatile (as time allows)

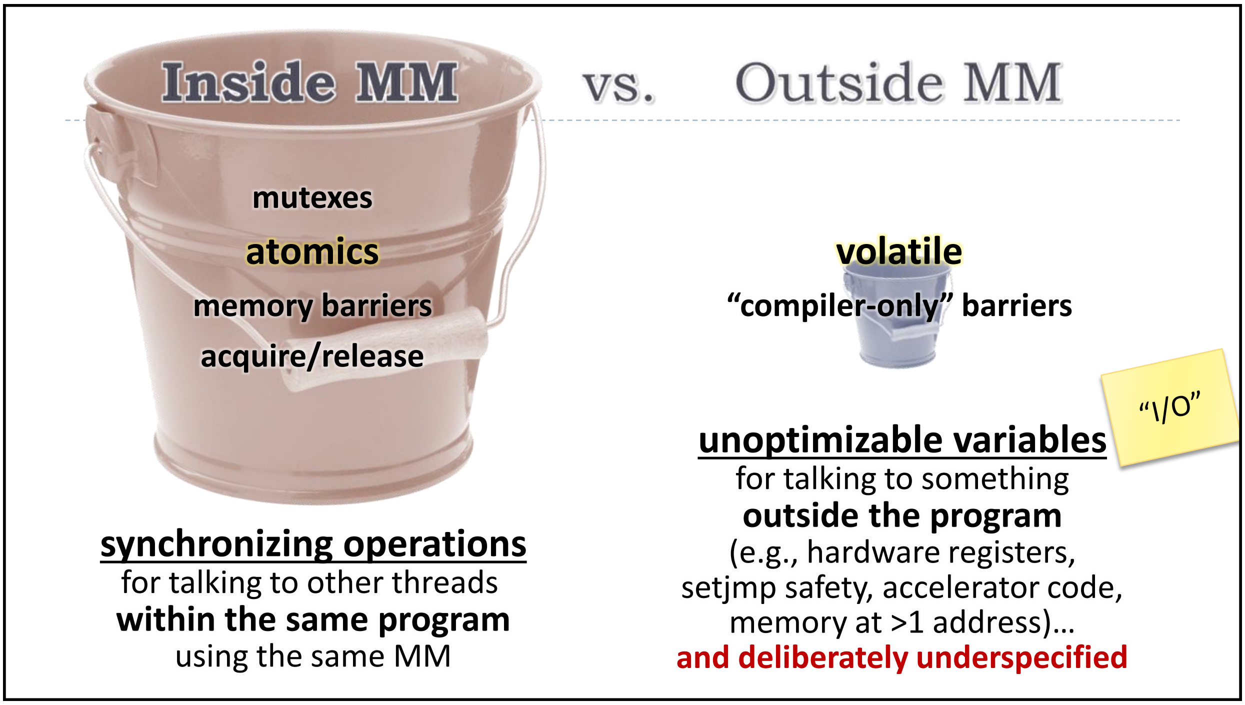


Fact of Life

volatile (Java, ≈.NET) **!=** **volatile** (C, C++)

==

atomic (C, C++)



Inside MM vs. Outside MM

mutexes

atomics

memory barriers

acquire/release

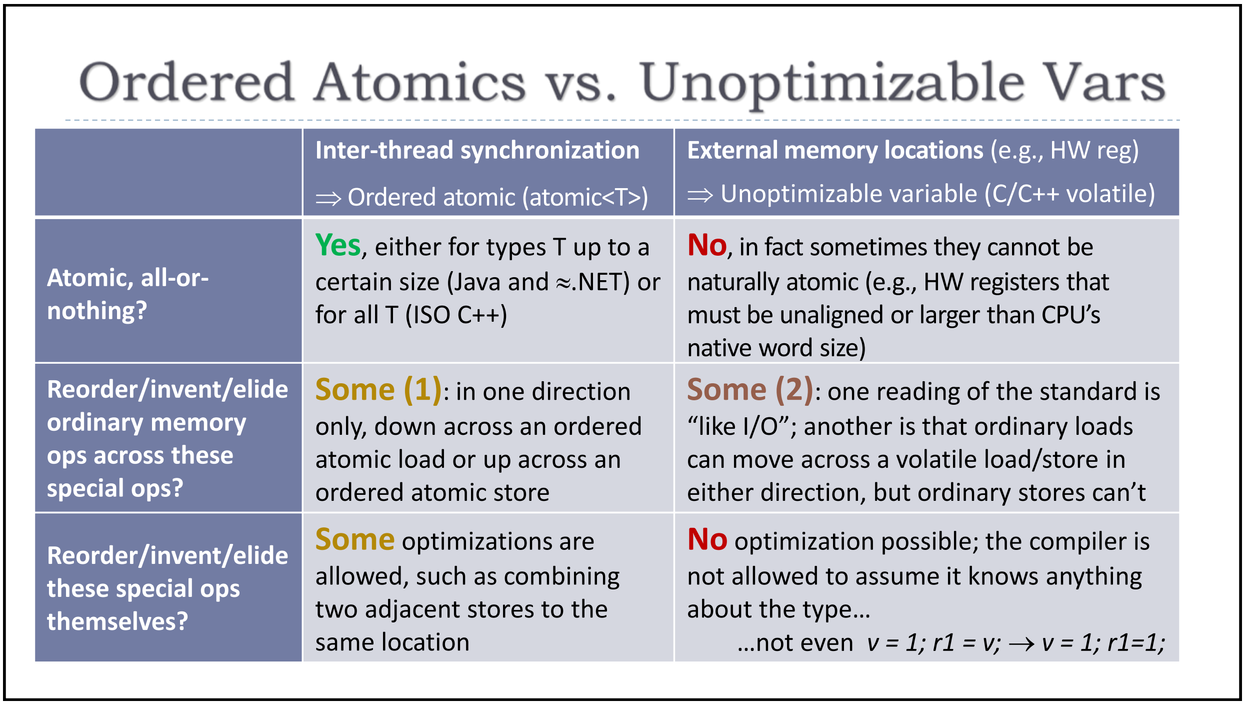
volatile

“compiler-only” barriers

“I/O”

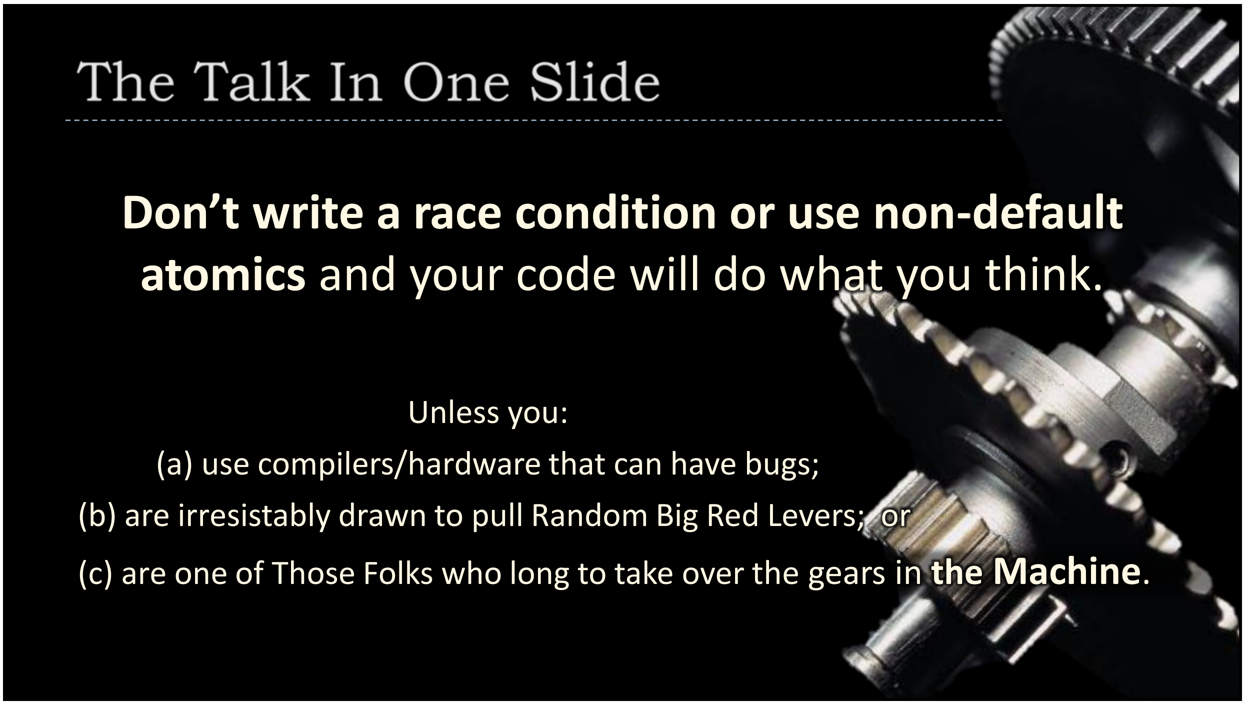
unoptimizable variables
for talking to something
outside the program
(e.g., hardware registers,
setjmp safety, accelerator code,
memory at >1 address)…
and deliberately underspecified

synchronizing operations
for talking to other threads
within the same program
using the same MM



Ordered Atomics vs. Unoptimizable Vars

	Inter-thread synchronization ⇒ Ordered atomic (atomic<T>)	**External memory locations** (e.g., HW reg) ⇒ Unoptimizable variable (C/C++ volatile)
Atomic, all-or-nothing?	**Yes**, either for types T up to a certain size (Java and ≈.NET) or for all T (ISO C++)	**No**, in fact sometimes they cannot be naturally atomic (e.g., HW registers that must be unaligned or larger than CPU's native word size)
Reorder/invent/elide ordinary memory ops across these special ops?	**Some (1)**: in one direction only, down across an ordered atomic load or up across an ordered atomic store	**Some (2)**: one reading of the standard is "like I/O"; another is that ordinary loads can move across a volatile load/store in either direction, but ordinary stores can't
Reorder/invent/elide these special ops themselves?	**Some** optimizations are allowed, such as combining two adjacent stores to the same location	**No** optimization possible; the compiler is not allowed to assume it knows anything about the type… …not even *v = 1; r1 = v;* → *v = 1; r1=1;*



The Talk In One Slide

Don't write a race condition or use non-default atomics and your code will do what you think.

Unless you:

(a) use compilers/hardware that can have bugs;

(b) are irresistably drawn to pull Random Big Red Levers; or

(c) are one of Those Folks who long to take over the gears in **the Machine**.

RECALL: The Truth

- Q: Does your computer execute the program you wrote?

True □

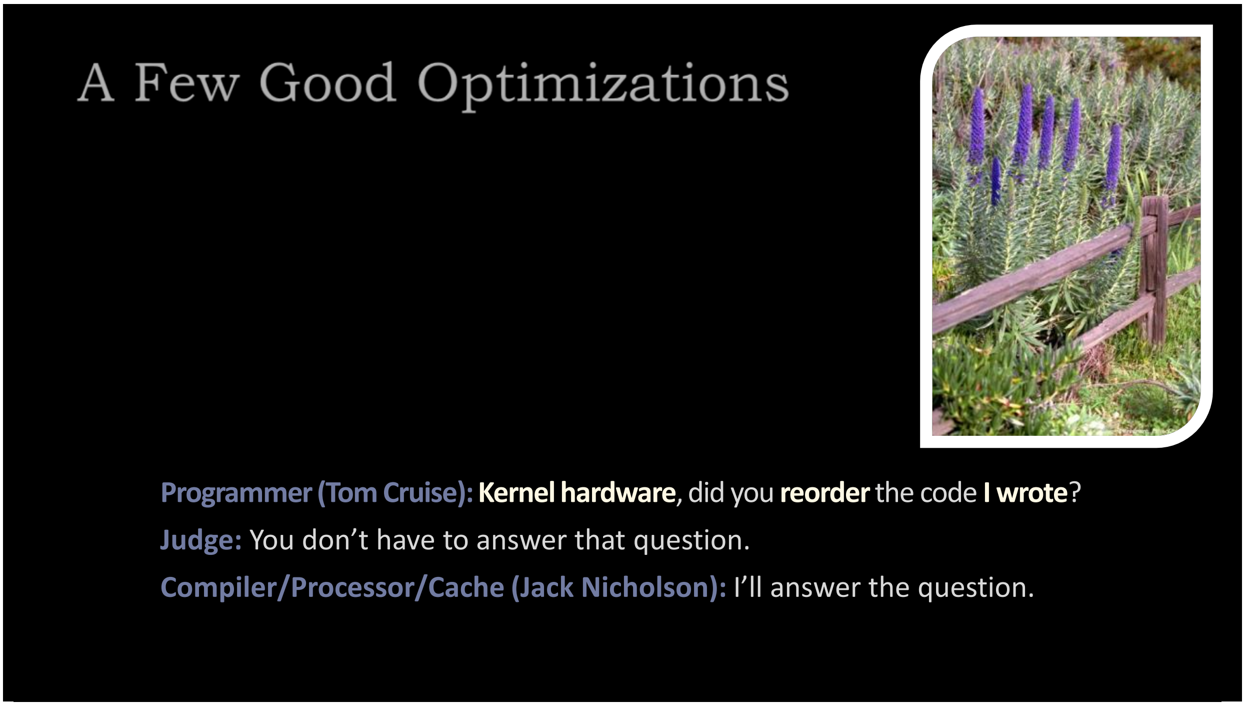
False ☑

RECALL: Sequential Consistency (SC)

- Hey, **sequential consistency (SC)** seems great!

  "… the result of any execution is the same as if the reads and writes occurred in some order, ***and the operations of each individual processor appear in this sequence in the order specified by its program****"*

- But chip/compiler designers can be annoyingly helpful:
  - It can be (much) more expensive to do exactly what you wrote.
  - Often they'd rather do something else, that could run (much) faster.
    - Common reaction: *"What do you mean, my program is too slow, you'll execute a different program instead…?!"*



A Few Good Optimizations

Programmer (Tom Cruise): **Kernel hardware**, did you **reorder** the code **I wrote**?

Judge: You don’t have to answer that question.

Compiler/Processor/Cache (Jack Nicholson): I’ll answer the question.