linux.conf.au 2008 Mel8ourne

After 25 Years, C/C++ Understands Concurrency

Paul E. McKenney, Distinguished Engineer
IBM Linux Technology Center

February 1, 2008

What This Talk is Not...

- **Not introducing new synchronization mechanisms**
  - The point of standardization is to codify ***existing*** practice
- **Not introducing new uses of or ways to test synchronization mechanisms**
  - The point of standardization is to codify ***existing*** practice
- **Not a comprehensive overview of c++0x**
  - For that see http://open-std.org/jtc1/sc22/wg21/docs/papers/

- **This talk is about concurrency features of c++0x, focused mainly on memory ordering**

Why C Programmers Should Care About C++...

- **Both C and C++ lack support for concurrency in the standard**
- **Both groups desire compatibility**
  - So C standards-committee members are participating in C++ concurrency standards effort
  - When C++ standard is complete, it will be adapted for C

How I Ended Up Messing With Standards...

- **Was working on a C++ project for the US Defense Advanced Research Projects Agency (DARPA)**
  - Embedded communications application based on Mach
  - Needed to influence the C++ standard due to shortcomings in the language
  - Heavily marked up a copy of C++ documentation
    - But the committee took it reasonably well

How I Ended Up Messing With Standards...

- **But wait... That was back in 1990!!!**
  - Fast-forwarding to 2005...

How I Ended Up Messing With Standards Again...

- **In May 2005, I hear rumors of C/C++ standardizing memory ordering models**
  - Not a surprise, as Java recently did the same
  - But quick Google search turns up nothing
  - Besides, was tearing hair out trying to implement realtime RCU
- **Fast-forward to late 2006...**

How I Ended Up Messing With Standards Again...

- **More persistent rumors surface**
  - Along with complaints that proposed standard favors Itanium
- **But this time the group was evident, including email list**
- **I joined the mailing list, planning to lurk for a few weeks**

What I Learned While Lurking...

- **Concurrency subgroup had high opinion of Linux:**
  - “So read_barrier_depends() stuff in Linux is also totally busted. (Just like refcounting, etc.)” (2005)
  - “And I don't believe that the semantics of read_barrier_depends() are actually definable” (2006)
- **I was only able to remain in lurk mode for about 3 days**
- **Though this high opinion persisted for some time:**
  - “And I think that does work for RCU, at least for conventional optimizations. But the more I think about, the less I'm convinced that it's 100% reliable.” (2007)

But Don't C/C++ Already Handle Concurrency???

- **And I *have* been doing parallel C for about 17 years**
- **But I have always used non-standard extensions**
  - Linux kernel uses non-standard asms for memory barriers, atomic operations, RCU, ...
  - Compiler writers generally don't worry about concurrency
    - “The standard says that the result is undefined!!! So I can do anything!!!”
    - Things can break easily...
    - Which might well explain the concurrency subgroup's skepticism!!!
- **So, what *real* problems can arise?**
  - Refetching variables (as in mce_log() needing rmb()...)
  - Fetch variables piece at a time, or merge stores
  - Stores clobber adjacent variables, compiler does additional stores to “fix things up” -- too bad if shared variables!!!
  - Re-order code (e.g., pulling critical section ahead of lock)
- **Can fix these, but requires constant attention**

Refetching Variables

- **Consider following code:**

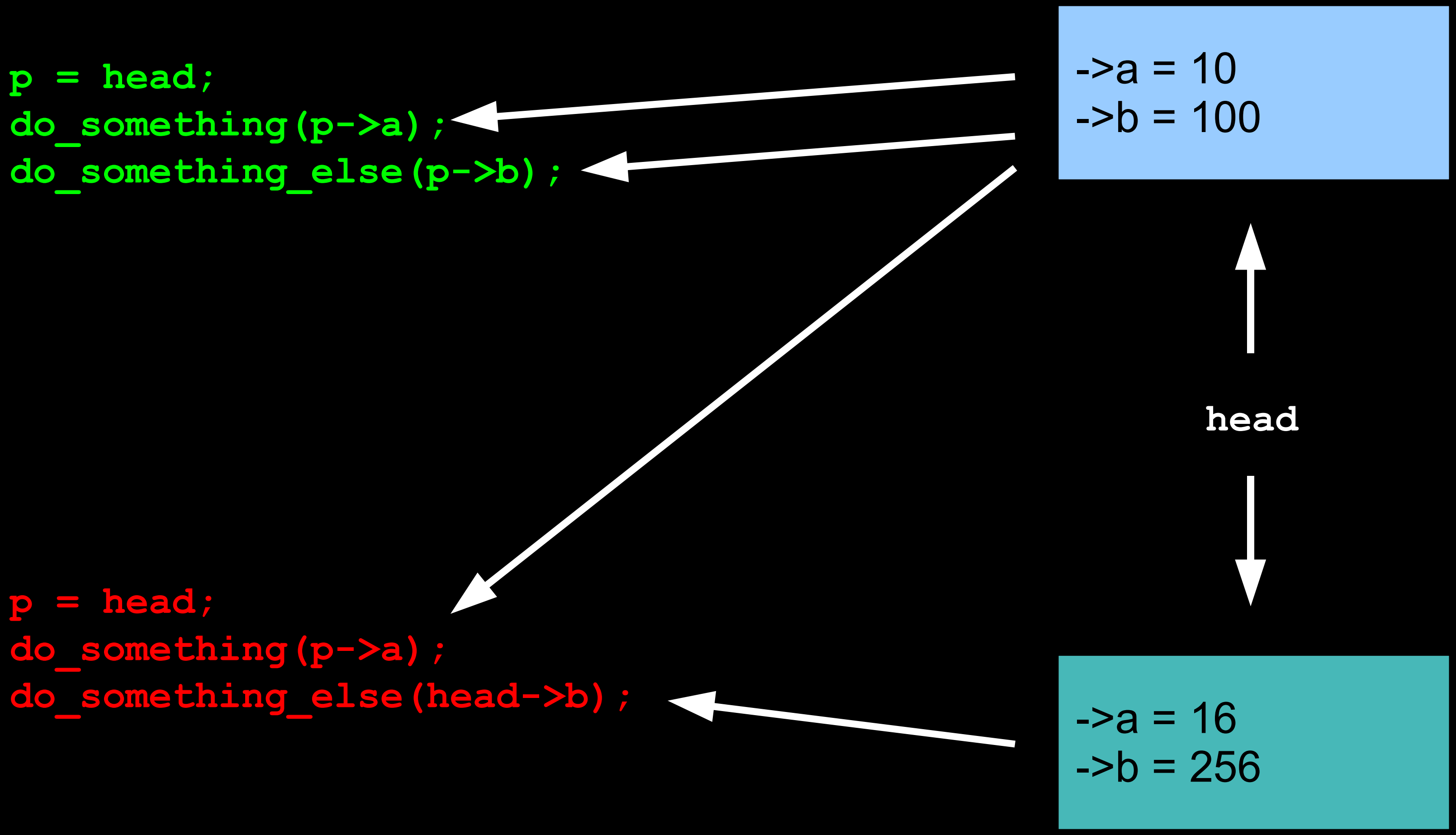
```
p = head;
do_something(p->a);
do_something_else(p->b);
```

- **Compiler might handle register pressure by refetching:**

```
p = head;
do_something(p->a);
do_something_else(head->b);
```

- **If some other task modified “head” in the meantime, the code might see inconsistent values**
- **Why???**
  - The compiler might run out of registers on some machines
  - Like the 32-bit x86...

Refetching Variables



```
p = head;
do_something(p->a);
do_something_else(p->b);
```

```
p = head;
do_something(p->a);
do_something_else(head->b);
```

Piece-at-a-Time Variable References

- **Consider following code:**

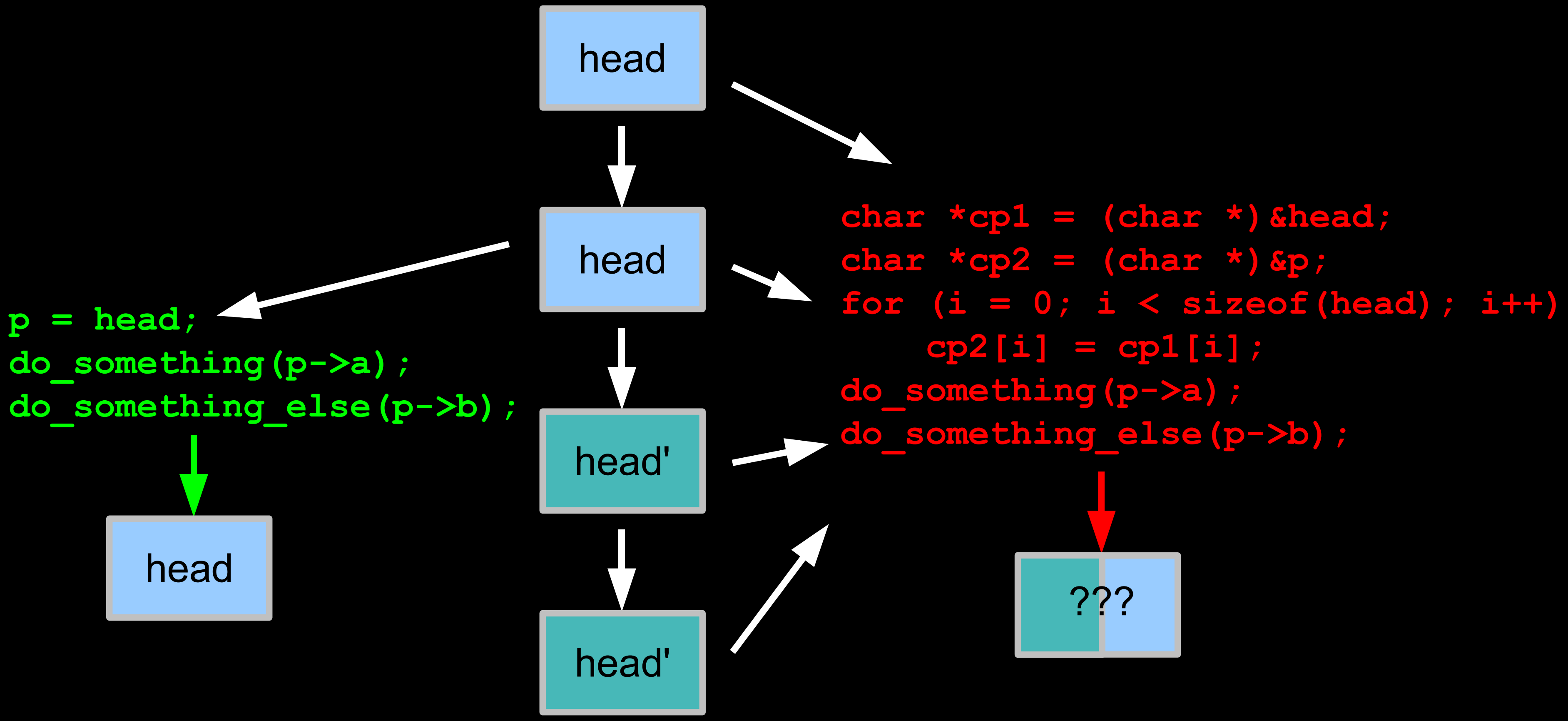
```
p = head;
do_something(p->a);
do_something_else(p->b);
```

- **Compiler might fetch piece-at-a-time:**

```
char *cp1 = (char *)&head;
char *cp2 = (char *)&p;
for (i = 0; i < sizeof(head); i++)
    cp2[i] = cp1[i];
do_something(p->a);
do_something_else(p->b);
```

- **If some other task modified “head” in the meantime, might see bitwise mash-up of the old and new values**
- **Why???**
  - Consider an 8-bit CPU, which the C language must handle
  - Fortunately, the Linux kernel prohibits such throwbacks

Piece-at-a-Time Variable Reference



```
p = head;
do_something(p->a);
do_something_else(p->b);
```

```
char *cp1 = (char *)&head;
char *cp2 = (char *)&p;
for (i = 0; i < sizeof(head); i++)
    cp2[i] = cp1[i];
do_something(p->a);
do_something_else(p->b);
```

Clobbering Adjacent Variables

- **Consider following code:**

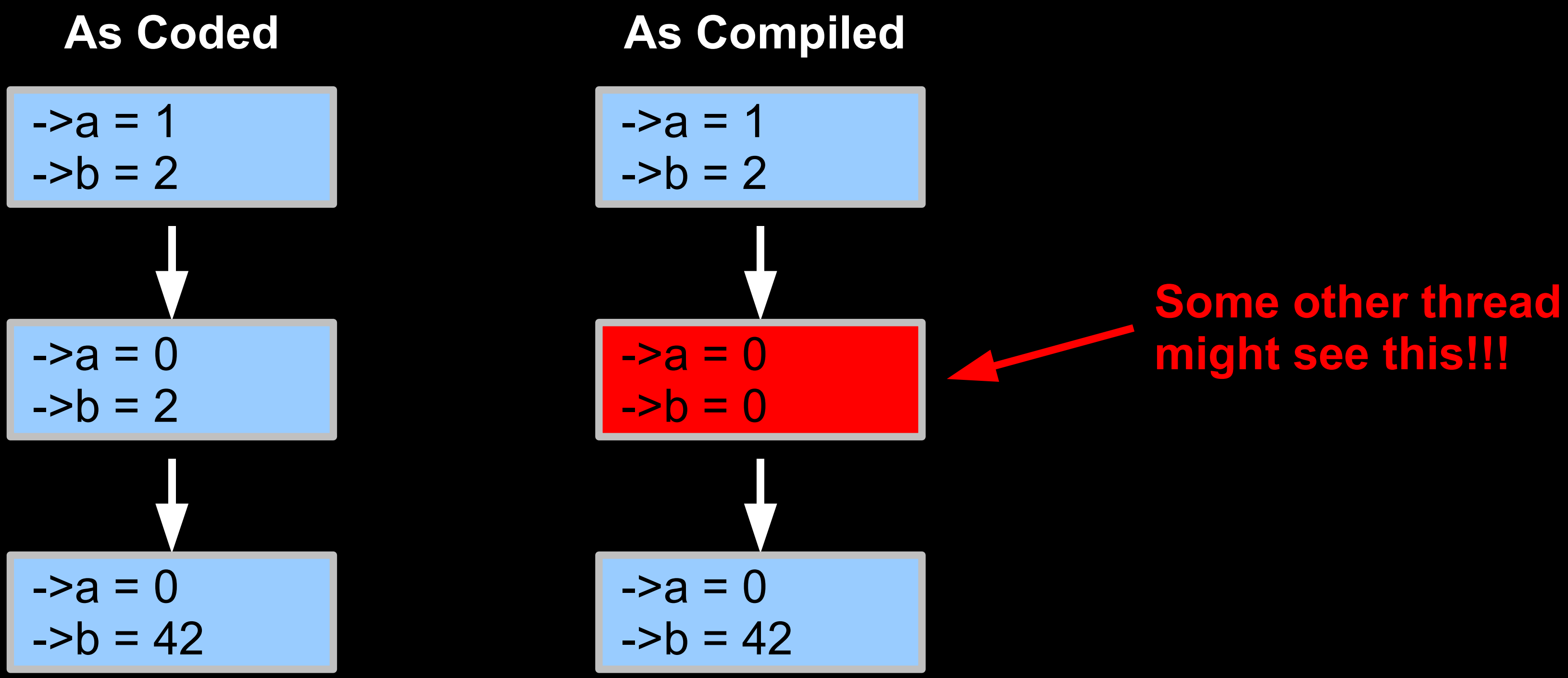
```
struct foo {
    short a, b;
} f = { 1, 2 };
f.a = 0;
f.b = 42;
```

- **Compiler might clobber whole structure:**

```
f = 0;
f.b = 42;
```

- **If some other task is watching, it might see f.b==0**
  - Despite the fact that this value logically never occurs!
- **Why???**
  - Consider a 32-bit CPU with expensive 16-bit memory references
    - Or some vector machines...

Clobbering Adjacent Variables



As Coded

->a = 1
->b = 2

->a = 0
->b = 2

->a = 0
->b = 42

As Compiled

->a = 1
->b = 2

->a = 0
->b = 0

->a = 0
->b = 42

Some other thread might see this!!!

The compiler assumes that there are no other threads!!!

Re-Ordering Code

- **Consider following code:**

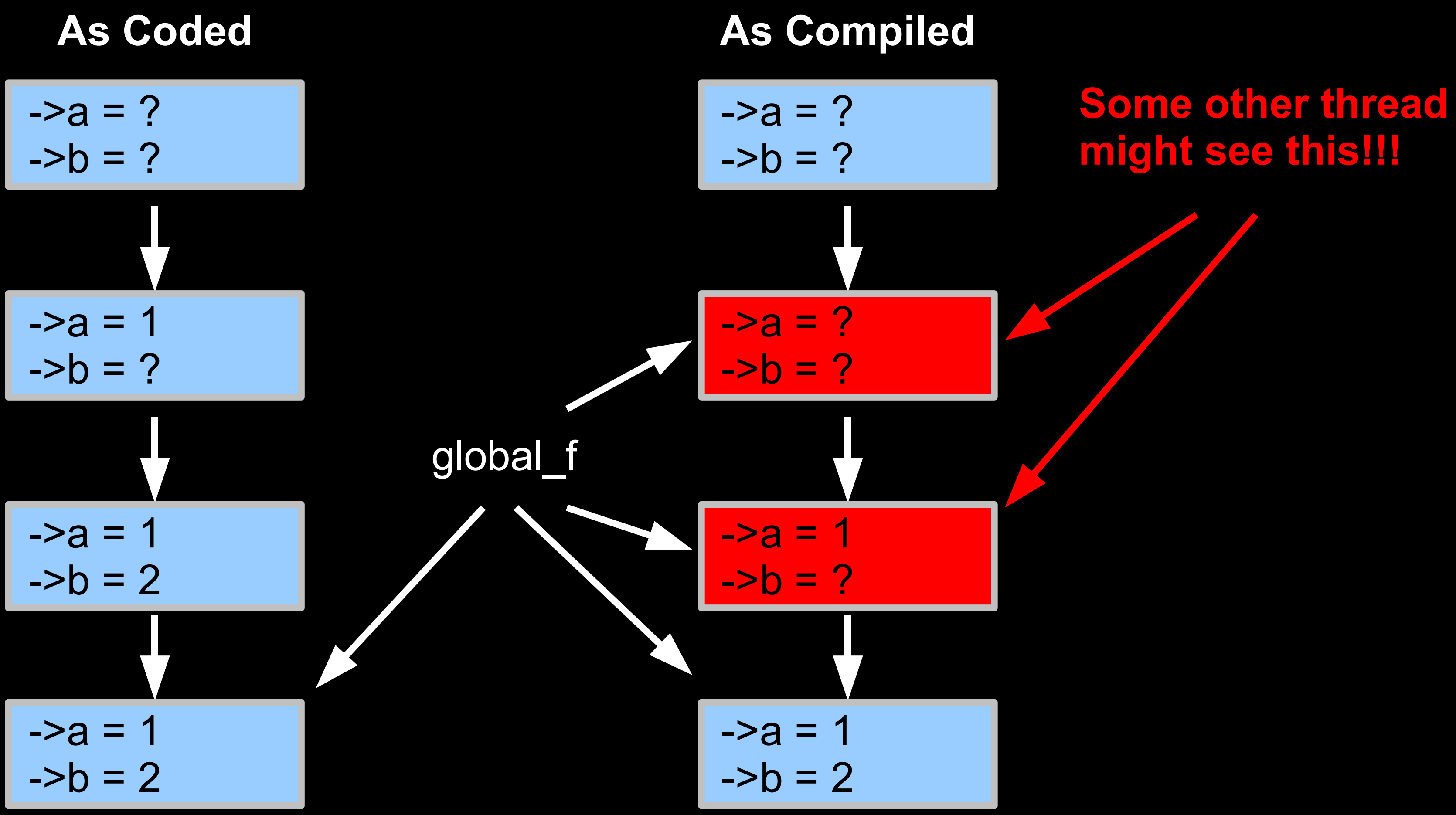
```
f.a = 1;
f.b = 2;
global_f = f;
```

- **Compiler might re-order assignments:**

```
global_f = f;
f.a = 1;
f.b = 2;
```

- **If some irq handler or some other task is watching, it might see uninitialized values for f.a and f.b!!!**
  - Linux kernel uses barrier() to prevent this (asm with “memory”)
  - But it is also necessary to prevent the CPU from reordering!
    - smp_mb() and friends
- **Why??? Consider a CPU with few registers...**
  - Like 32-bit x86...

Re-Ordering Code



The compiler assumes that there are no other threads!!!

Why Does C/C++ Allow Such Things???

- **Optimization, performance, existing compilers, strange CPUs (8-bit CPUs, CPUs with no byte operations, ...)**
  - Approach: define “atomic” type restricting optimizations
  - Sort of like “volatile”, but with well-defined semantics in multi-threaded environments
  - Non-atomic variables undefined in presence of “data races”
    - Where at least one thread updates concurrently with other threads accessing—protect non-atomic variables with locks, &c
- **Logical next step would be to define memory barriers**
  - However, this proved surprisingly controversial
  - Though not without reason: the Linux community is not the only group who find the semantics of memory barriers to be rather obscure

What Does One Use Instead of Memory Barriers?

- **Store-release and load-acquire on a variable**
  - My initial reaction: “What do you think you are doing attempting to write Itanium instructions into the standard???”

What Does One Use Instead of Memory Barriers?

- **Store-release and load-acquire on a variable**
  - My initial reaction: "What do you think you are doing attempting to write Itanium instructions into the standard???"
  - To be fair, I suspect that a few other members were concerned that I was attempting to write IBM's RCU patents into the standard
    - I (just barely) resisted the temptation to point out that the first RCU patents are likely to expire before highly reliable compilers conforming to the new c++0x standard see the light of day
    - I instead pointed out that garbage collectors (is in progress for C++), hazard pointers, or type-safe memory could take the place of RCU

Does Store-Release and Load-Acquire Work?

- **Store-release and load-acquire work nicely on all parallel architectures, including POWER**
  - Prohibits all reorderings except the important store-buffer store-load case, permitting light-weight barrier instructions:
    - x86: nothing (given new Intel and AMD memory models)
    - POWER/PowerPC: lwsync for store, bc;isync for load
    - Itanium: ld,acq & st,rel
    - s390: nothing
- **Store-release and load-acquire easy (easier) to explain**
  - Store-release is “publish” operation for prior stores
  - Load-acquire is “subscribe” operation for later accesses
    - Which are guaranteed to see stores published by the store-release
- **Roughly half of Linux smp_mb() convert trivially**
  - Others might require more work
  - But would likely make the code much easier to understand

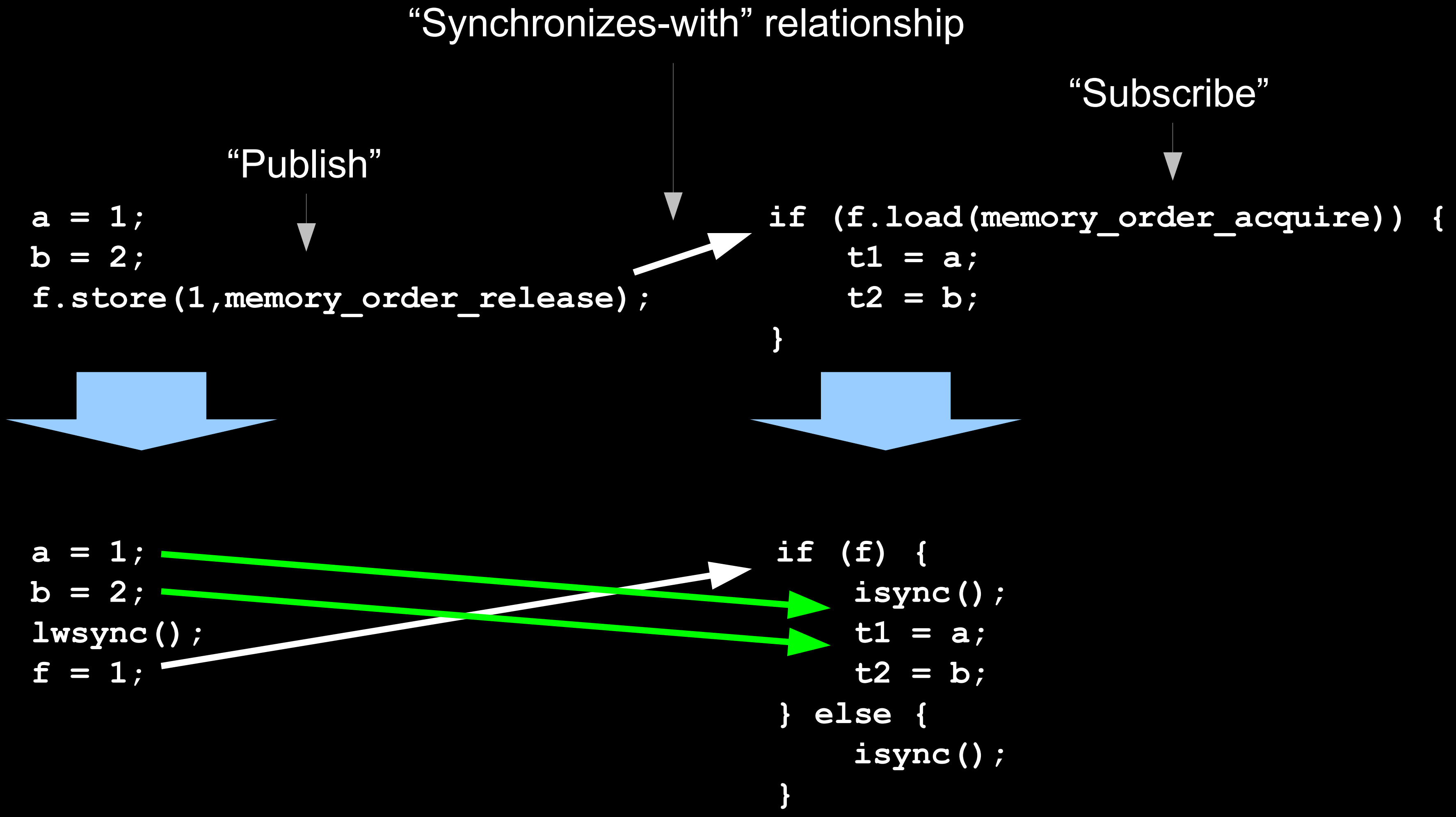
Store-Release and Load-Acquire Semantics

		Before Barrier	
		LOAD	STORE
After Barrier	LOAD	Ordered	Unordered
	STORE	Ordered	Ordered

```
f.store(1,memory_order_release);
/* Subsequent loads may be reordered to precede f.store() */
/* Subsequent stores as well (by the compiler and Itanium) */

/* Prior stores may be reordered to follow f.load() */
/* Prior loads as well (by the compiler and Itanium). */
r1 = f.load(memory_order_acquire);
```

Store-Release and Load-Acquire Example: POWER



“Synchronizes-with” relationship

“Subscribe”

“Publish”

```
a = 1;
b = 2;
f.store(1,memory_order_release);
```

```
if (f.load(memory_order_acquire)) {
    t1 = a;
    t2 = b;
}
```

```
a = 1;
b = 2;
lwsync();
f = 1;
```

```
if (f) {
    isync();
    t1 = a;
    t2 = b;
} else {
    isync();
}
```

Can We Dispense With Raw Memory Barriers?

- **Probably not, though many committee members tried**
  - Large amount of existing usage, corner cases
  - Bjarne Stroustrup had to intervene to keep memory-barriers
    - Existing software? Who cares about existing software???
- **Some distributed-shared-memory folks *hate* barriers!!!**
  - Some distributed-memory guys use variable as “tag”
    - Idea seems to be to ship groups of variables instead of pages, limiting communications intensity
    - Not all distributed-memory people see this as a critical issue
      - Unfortunately, this group was not represented on the committee
  - After some debate, invented mythical global variable with very long name for unadorned memory barriers
    - Which conventional machines are free to ignore and which programmers never have to type in
    - Except those compiling for such distributed-memory machines

What Should C/C++ Memory Model Be?

- **Theoretical group wanted sequential consistency (SC)**
  - All operations on atomics globally ordered
  - On POWER, sync between all pairs of references to atomics
    - lwsync in some cases, but still expensive
  - See next slide for list of real-world use cases requiring SC
- **Committee-style compromise:**
  - SC is default for atomic variables
  - Weaker operations are available, including “relaxed” access that has no memory-ordering semantics
  - It will likely be possible to relax SC semantics in practice – but theory of near-SC still quite immature
- **Semi-formal semantics finalized**
  - Except for data-dependency ordering, which is still in progress
  - Despite a very rocky start...

Real-World SC Use Cases

Atomic Operations

- **Atomic operations**
  - If CPU does not support atomic operation, auto-generate locking
    - For example, compare-and-swap (AKA cmpxchg) on a large struct
    - Each type has a flag stating whether it is natively atomic
  - C++ templates used for atomic operation definitions
  - Can select degree of memory ordering desired
    - memory_order_relaxed: no ordering
    - memory_order_depends: dependency ordering (proposed)
    - memory_order_acquire: “acquire” ordering
    - memory_order_release: “release” ordering
    - memory_order_acq_rel: both “acquire” and “release” ordering
    - memory_order_seq_cst: full ordering with all seq_cst operations across all CPUs
  - Numerous operations: load, store, arithmetic, boolean, compare-and-swap, ...
- **Use of atomic variables in signal handlers**
  - But only atomic variables of primitive types!!!
  - (Current restriction is sig_atomic_t)

Other Concurrency Features Being Considered

- **Boost.Threads library functions**
  - Threads, mutexes/locks, condition variables, call-once functions
  - Thread cancellation caused much debate: strange interactions with destructors & exception handlers in some implementations
    - Voluntary cancellation particularly problematic
  - Garbage collection (proposed)
- **Some complications:**
  - Destructors running concurrently with constructors for same object
  - Destructors running concurrently with exit() or atexit() handlers
    - Simplification: terminate all threads before exiting!!!
    - New quick_exit() exits without executing destructors (but invokes at_quick_exit() handlers)
      - And at_quick_exit() handlers can register at_quick_exit() handlers...
  - Code relying on destructors running in reverse order of constructors
  - Garbage collector with finalization

What Does All This Mean For F/OSS?

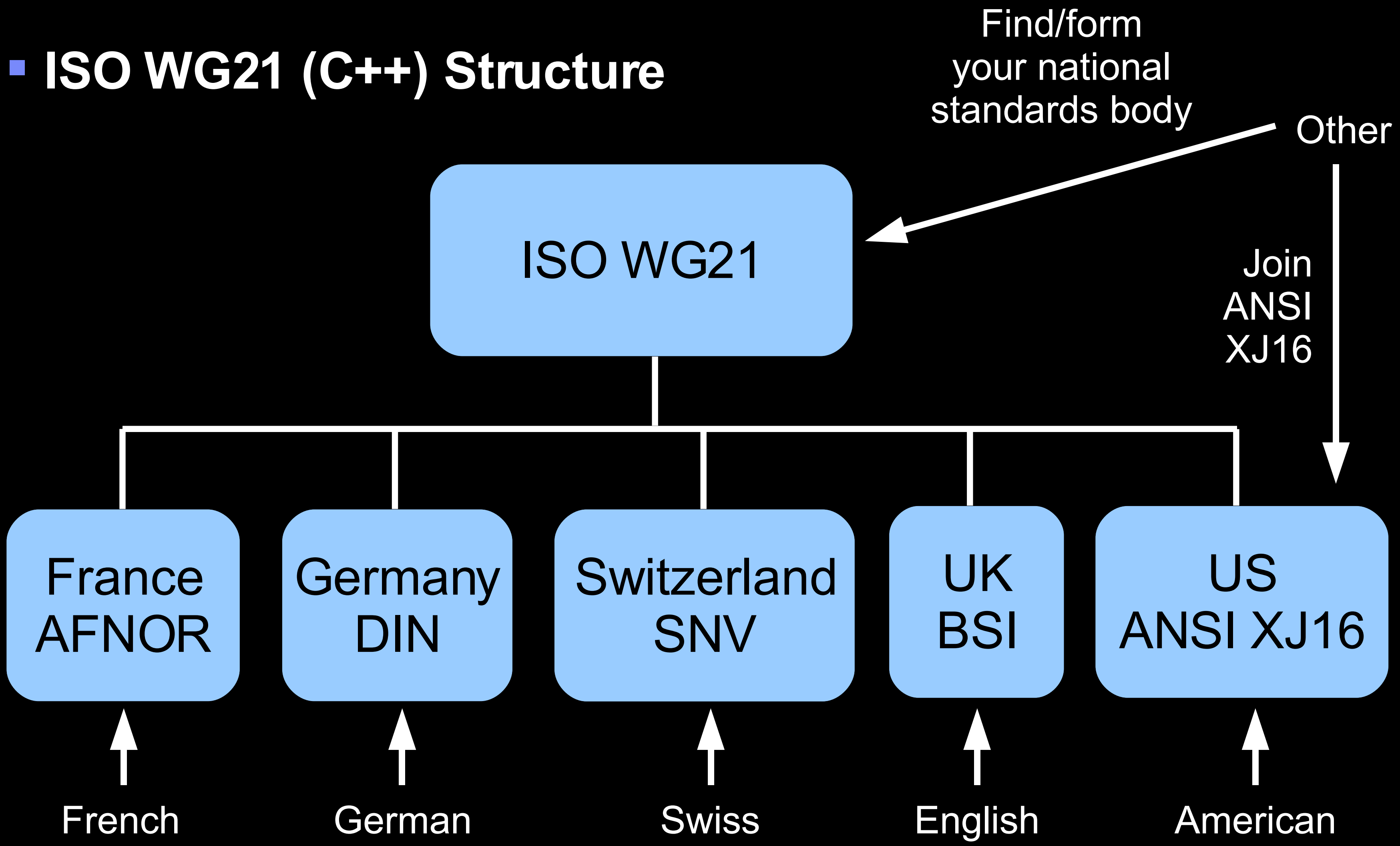
- **Multithreaded software actually favors F/OSS!!!**
  - Multithreaded SW requires global design constraints
    - Deadlock avoidance
    - Data structure partitioning
    - Reducing lock and memory contention
  - F/OSS “shows you the code”, allowing any developer to verify global design constraints
    - Also works in tightly controlled proprietary environments
    - But not given mutually proprietary plug-ins sharing the same address space
      - Same problem that is posed by Linux-kernel binary modules/drivers!!!
- **There is potential to move low-level concurrency code from system.h, atomic.h, &c to the compiler**
  - The compiler might be able to generate better code given the association with variables
  - Balanced by the fact that c++0x takes a different approach than do most existing projects...

Lessons Learned

- **Get involved early (see next slide)**
  - Though in this case, more-recent theoretical work on RCU was critically important
    - Early-2005 RCU nomenclature probably have not been convincing
  - But starting in 2005 might have produced an alternative to sequential consistency
  - Fortunately for me, a number of Linux-community members have been involved for quite some time ☺
- **Academia is important**
  - People listen to academics, even when we practitioners think that they shouldn't ☺
- **When standards people say “it is undefinable”, they sometimes really mean “I don't understand it”.**

How You Can Get Involved

- **ISO WG21 (C++) Structure**



Find/form your national standards body

Other

Join ANSI XJ16

ISO WG21

France AFNOR

Germany DIN

Switzerland SNV

UK BSI

US ANSI XJ16

French

German

Swiss

English

American

Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM, IBM (logo), e-business (logo), pSeries, e (logo) server, and xSeries are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.

Questions?