C++11/14/17 atomics and memory model: Before the story consumes you (Followed by Paul Mckenney's C++ Atomics: The Sad Story of memory_order_consume *A Happy Ending At Last?* **)**

Michael Wong
michaelw@ca.ibm.com
IBM Toronto Lab

International Standard Trouble Maker, Chief Cat Herder
IBM and Canadian C++ Standard Committee HoD
OpenMP CEO
Chair of WG21 SG14 Games dev/Low Latency/Financial/Trading
Chair of WG21 SG5 Transactional Memory
Director, Vice President of ISOCPP.org
C/C++ Standard
Vice Chair Standards Council of Canada Programming Languages

Acknowledgement and Disclaimer

- Numerous people internal and external, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk.
- Special thanks to Paul Mckenney and permission for images.
- Special call out to Anthony Williams whose book C++ Concurrency in Action was the first book that explained it all
- I even lifted this acknowledgement and disclaimer from some of them.
- But I claim all credit for errors, and stupid mistakes. **These are mine, all mine!**
- Any opinions expressed in this presentation are my opinions and do not necessarily reflect the opinions of IBM.

Legal Disclaimer

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM, PowerPC and the IBM logo are trademarks or registered trademarks of IBM or its subsidiaries in the United States and other countries.
- Other company, product, and service names may be trademarks or service marks of others..

IBM (IR)Rational Disclaimer

- © Copyright IBM Corporation 2015 All rights reserved. The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way. IBM, the IBM logo, Rational, the Rational logo, Telelogic, the Telelogic logo, and other IBM products and services are trademarks of the International Business Machines Corporation, in the United States, other countries or both. Other company, product, or service names may be trademarks or service marks of others.

Agenda

- Why are you attending this talk?
- Multithreading support in C11 and C++11
- Memory Model, the gory details

Keep it simple:
as simple as possible,
but no simpler.
– A. Einstein

Why are you here

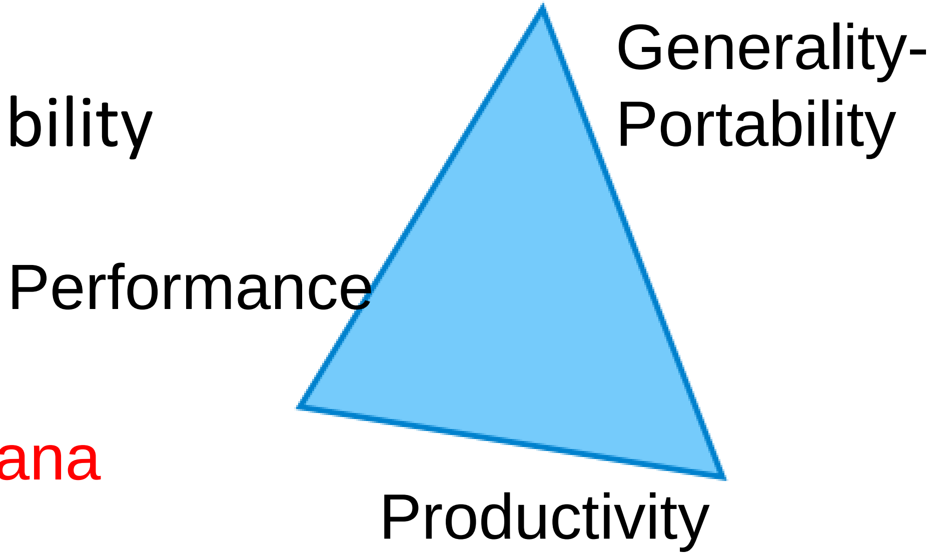
- You want things to go fast
- You want things to be productive
- You want things to be portable
- But You also want things to be correct

Why am I here

- Motivate why we have a memory model
- Teach the memory model (at least the first 5 consistency orders)
- Show use cases
- Performance is the main reason why we have a memory model; otherwise why do parallel programming?
  - Correctness is important
  - Can we still do parallel programming, make it correct but do is easier?
    - Yes, bring the tools to where it is
    - Teach them what it really means
- C++ Memory model goal:
  - Maximum performance but still maintain portability

So What are the Goals?

- Goals of Parallel Programming over and above sequential programming
  1. Performance
  2. Productivity
  3. Generality-Portability



The Iron Triangle of Parallel Programming language nirvana

- Oh, really??? What about correctness, maintainability, robustness, and so on?

- And if correctness, maintainability, and robustness don’t make the list, why do productivity and generality?

Agenda

- Why are you attending this talk?
- Multithreading support in C11 and C++11
- Memory Model, the gory details

Keep it simple:
as simple as possible,
but no simpler.
– A. Einstein

Is this valid C++ today? Are these equivalent?

```
int x = 0;
atomic<int> y = 0;
```

Thread 1:

```
   x = 17;
   y.store(1,
   memory_order_release);
   // or:      y.store(1);
```

Thread 2:

```
   while
   (y.load(memory_order_acquire
   ) != 1)
   // or:      while (y.load()
   != 1)

   assert(x == 17);
```

Yes, they are valid!

```
int x = 0;
atomic<int> y = 0;
```

Thread 1:

```
   x = 17;
   y = 1;
```

Thread 2:

```
   while (y != 1)
          continue;
   assert(x == 17);
```

What do you mean equivalent? Do you mean they behave the same? Or have the same performance?

Concurrency in C11/C++11/C++14/C++17

- C99/C++98/03: does not have concurrency
- C++11 is in Final Draft International Standard on 2011
- C11 is in Draft International Standard in 2011
- C++11 have multithreading support
  - Memory model, atomics API
  - Language support: TLS, static init, termination, lambda function
  - Library support: thread start, join, terminate, mutex, condition variable
  - Advanced abstractions: basic futures, thread pools
- C11 will have similar memory model, atomics API. TLS, static init/termination
  - Some minor differences like __Atomic qualifier
- C++14 enhanced the memory model with better definition of lock-free vs obstruction-free
  - Clarified atomic_signal_fence
  - Improved on out-of-thin-air results
  - Realized consume ordering as described was problematic, hence Paul McKenney's work to improve on it (see his talk)
- C++17 we are talking about
  - Define strength ordering of memory models
  - Execution agents, atomic views, better SC model

Memory Model and Consistency model, a quick tutorial

- Sequential Consistency (SC)

  Sequential consistency was originally defined in 1979 by Leslie Lamport as follows:

- “… 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 more expensive to do exactly what you wrote.
- Often they’d rather do something else, that could run faster.

Sequential Consistency: a tutorial

- The semantics of the single threaded program is defined by the program order of the statements. This is the strict sequential order. For example:

```
x = 1;
r1 = z;
y = 1;
r2 = w;
```

Sequential Consistency for program understanding

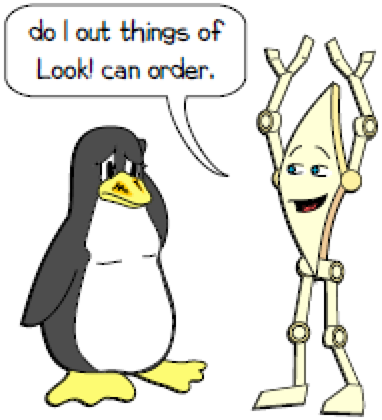
- Suppose we have two threads. Thread 1 is the sequence of statement above. Thread 2 is:

```
Thread 1:        Thread2:
x = 1;           w=1;
r1 = z;          r3=y;
y = 1;           z=1;
r2 = w;          r4=x;
```

(All variables are initialized to zero.)

- 2 of 4! Possible interleavings:

```
x = 1;           x=1;
w = 1;           w=1;
r1 = z;          r3=y;
r3 = y;          z=1;
y = 1;           r4=x;
z = 1;           r1=z;
r2 = w;          y=1;
r4 = x;          r2=w;
```



Now add fences to control reordering

Thread 1:

x = 1;

r1 = z;

fence();

y = 1;

r2 = w;

Is r3==1 and r4==1 possible?

Is r1==1 and r2==1 possible?

Thread2:

w=1;

r3=y;

fence();

z=1;

r4=x;

Memory Model and instruction reordering

- Definitions:
- Instruction reordering: When a program executes instructions, especially memory reads and writes, in an order that is different than the order specified in the program's source code.
- Memory model: Describes how memory reads and writes may appear to be executed relative to their program order.
- Affects the valid optimizations that can be performed by compilers, physical processors, and caches.

Agenda

- Multithreading support in C11 and C++11
- The problems of Concurrency before C/C++ 11
- Memory Model

Keep it simple:
as simple as possible,
but no simpler.
– A. Einstein

What is a memory model?

- A set of rules that describe allowable semantics for memory accesses on a computer program
  - – Defines the expected value or values returned by any loads in the program
  - – Determines when a program produces undefined behavior (e.g load from uninitialized variables)
- Stroustrup: represents a contract between the implementers and the programmers to ensure that most programmers do not have to think about the details of modern computer hardware
- critical component of concurrent programming

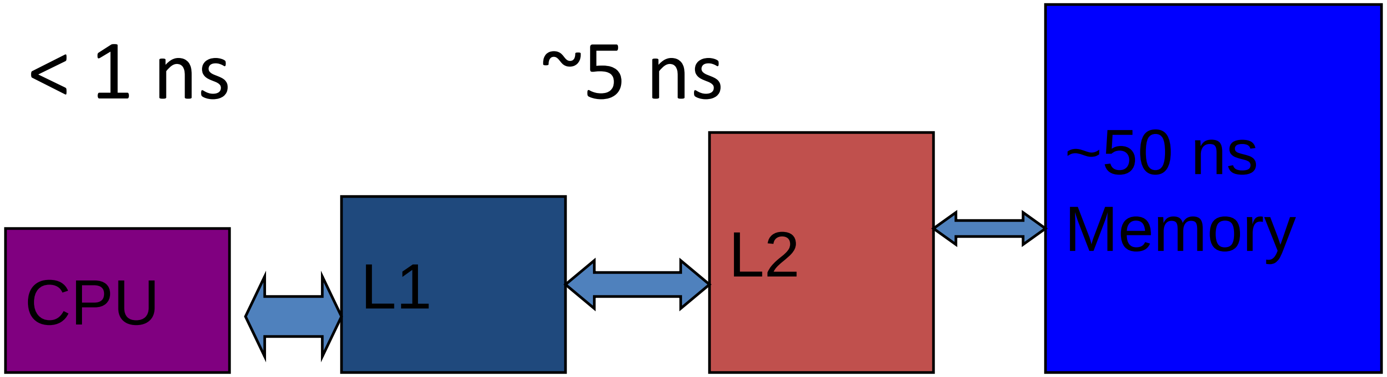
Thread switching and you

- Execution of a sequential program
- Software, not hardware
- A processor can run a thread
- Put it aside
  - Thread does I/O
  - Thread runs out of time
- Run another thread

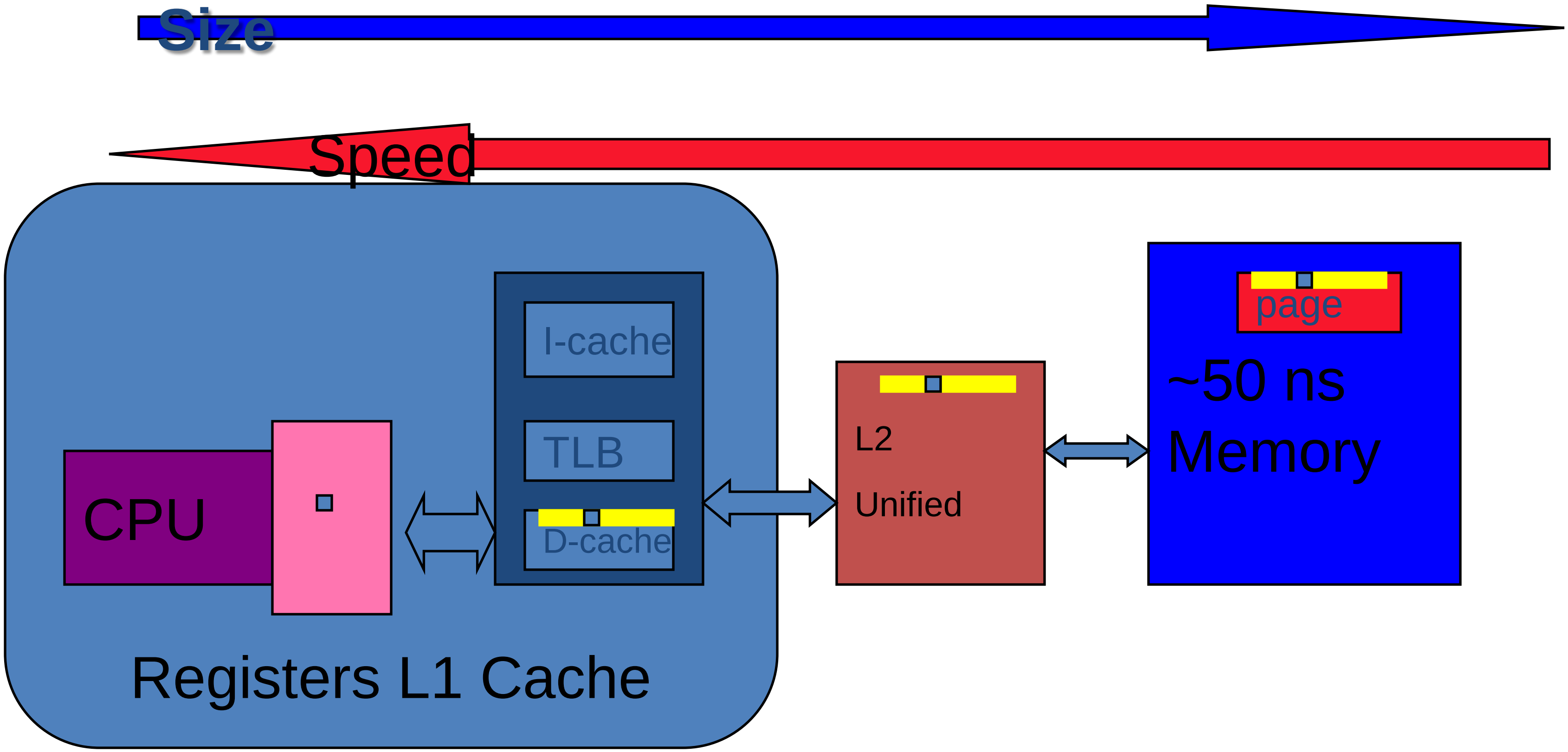
- You work in an office
- When you leave for lunch, someone else takes over your office.
- If you don’t take a break, a security guard shows up and escorts you to the cafeteria.
- When you return, you may get a different office

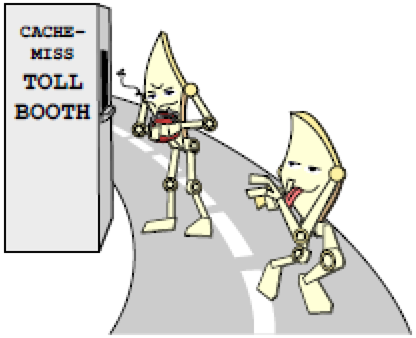
Memory vs CPU

- Memory latency is crucial to getting right performance, and affects parallelization
- Knowing the memory characteristics will help to write faster code, design language
- 1 GHz means one result every nanosecond
- Memory has latency 50 ns

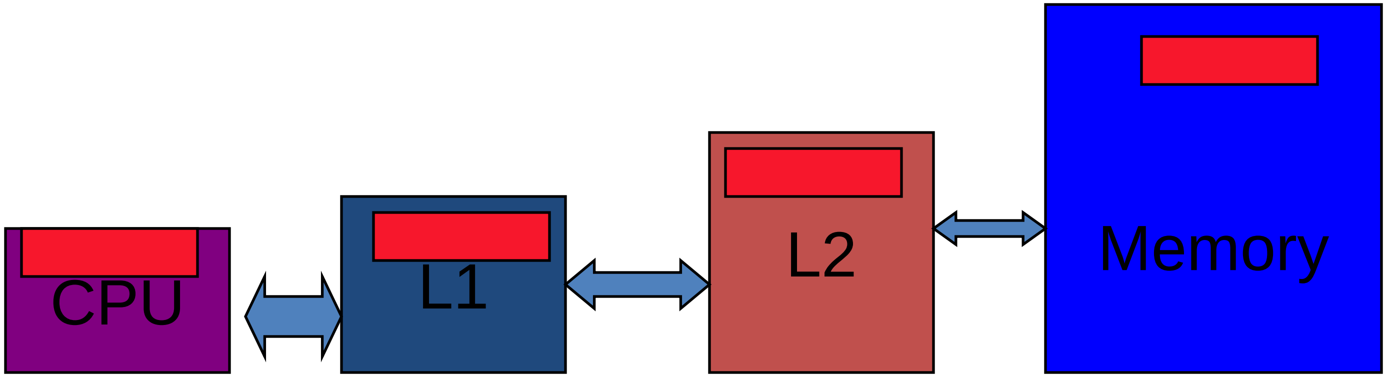


The memory hierarchy





Caches and Memory



CPU

L1

L2

Memory

1x

10-20x

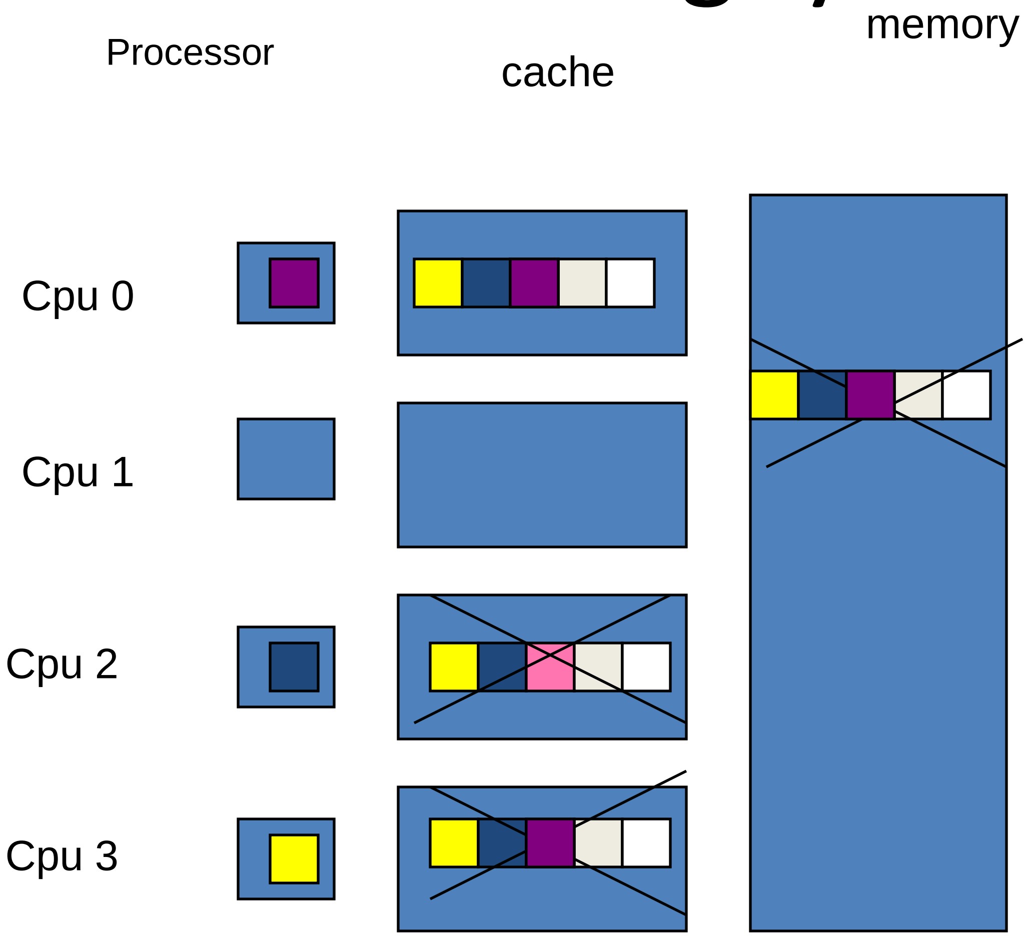
200x

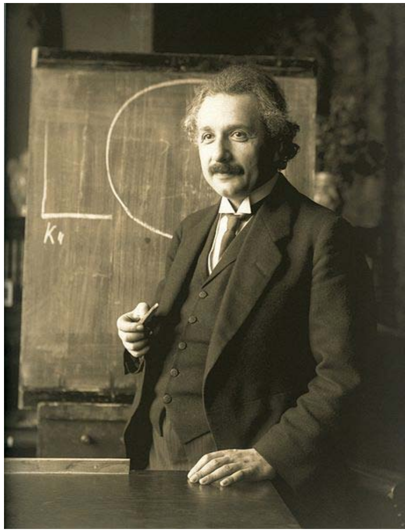
<1 ns

10-20 ns

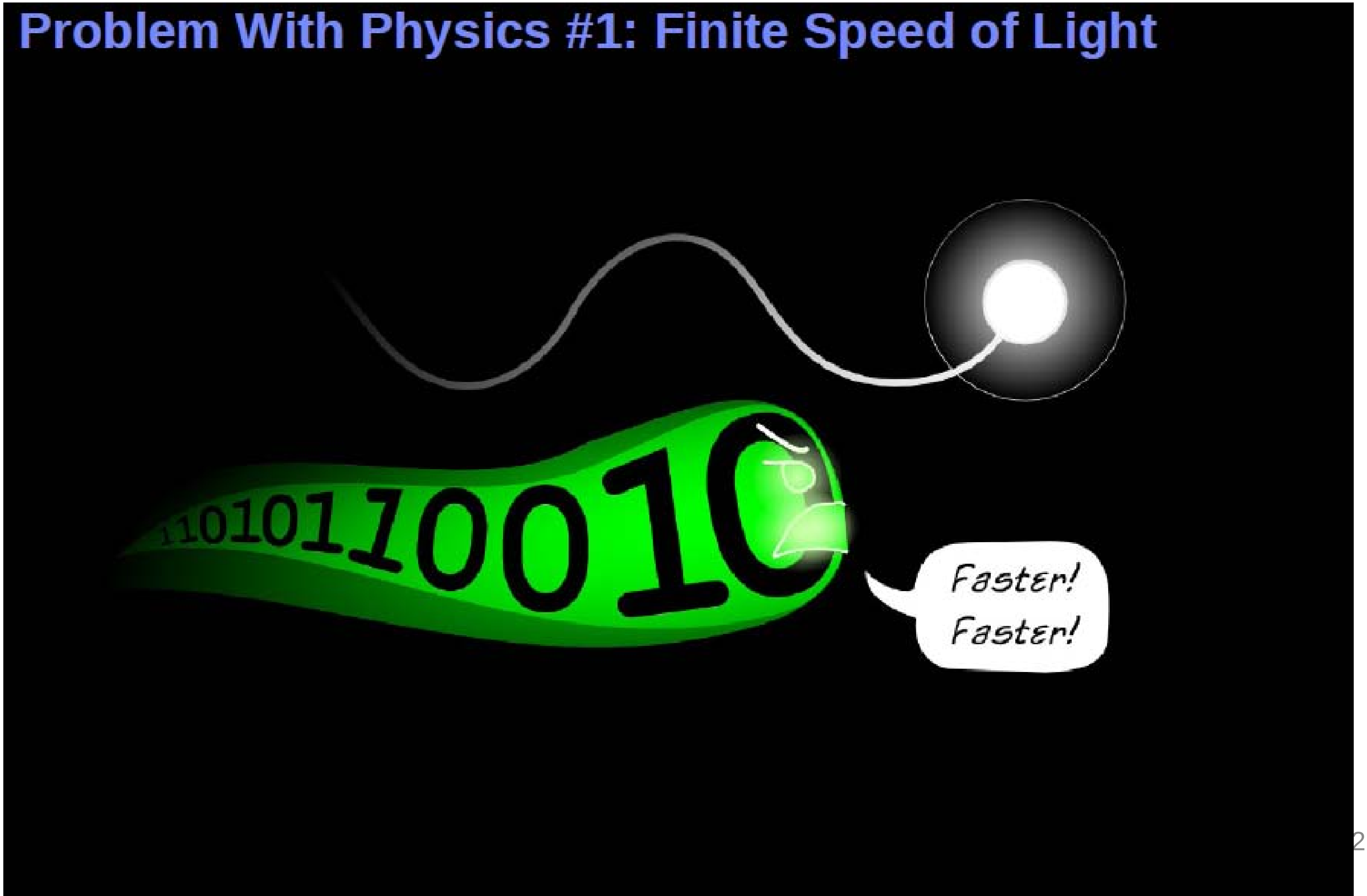
50-200 ns

Caches in an MP system and why do these guys hate us?



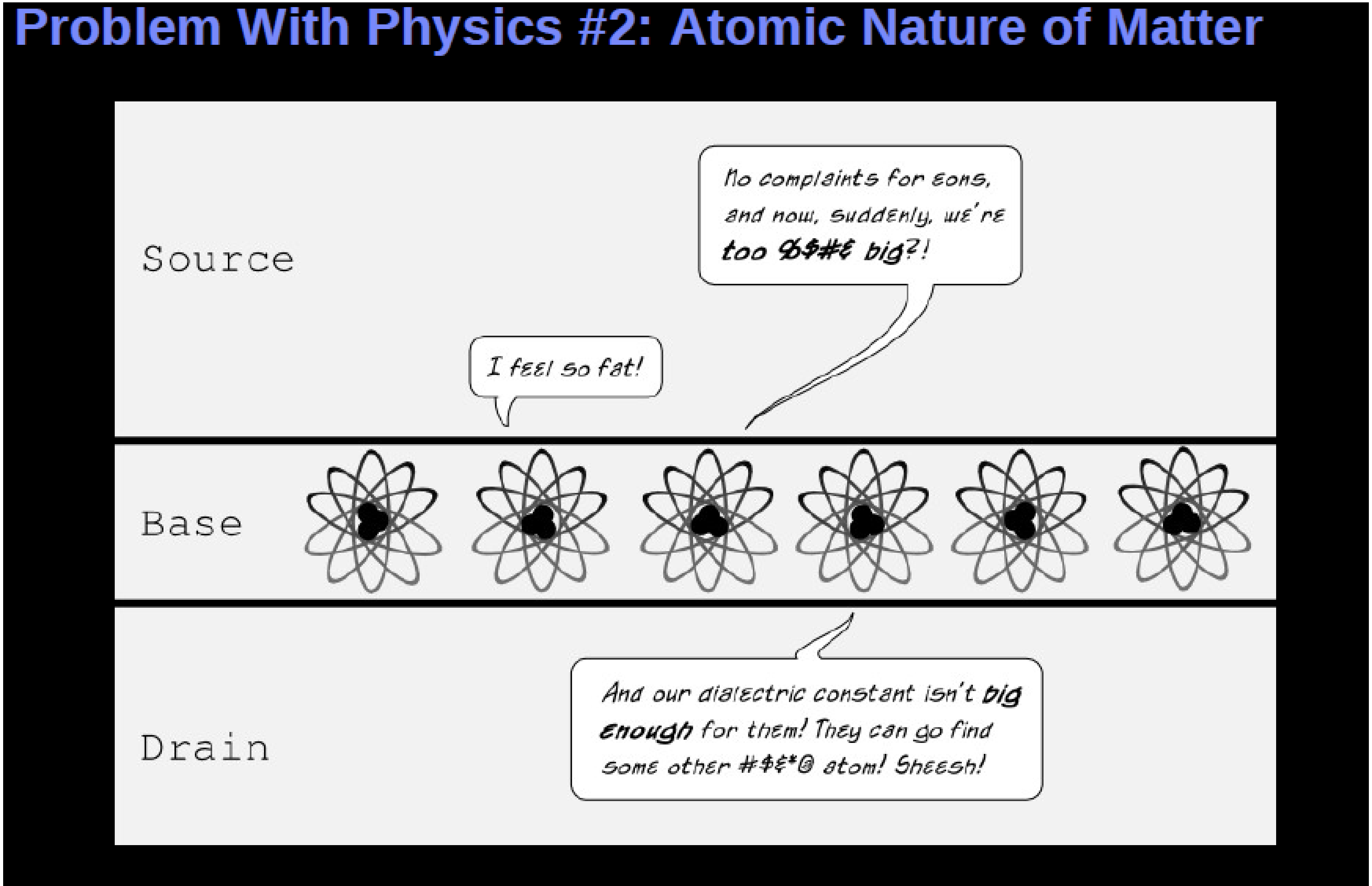


Problem With Physics #1:finite speed of light



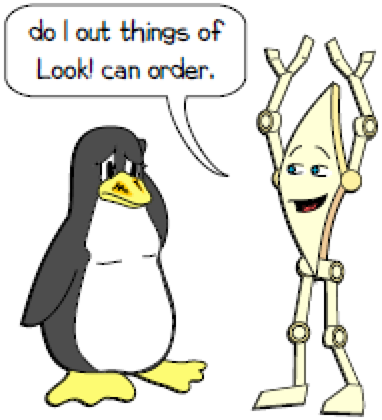
Problem With Physics #1: Finite Speed of Light

Problem with Physics #2: atomic nature of matter


Problem With Physics #2: Atomic Nature of Matter
Source
Base
Drain
No complaints for eons, and now, suddenly, we're too %$#€ big?!
I feel so fat!
And our dialectric constant isn't big enough for them! They can go find some other #$€*@ atom! Sheesh!

Memory Model

- One of the most important aspect of C++0x /C1x is almost invisible to most programmers
  - memory model
    - How threads interact through memory
    - What assumptions the compiler is allowed to make when generating code
    - 2 aspects
      - How things are laid out in memory
      - What happens when two threads access the same memory location and one of them is a modfy
        - Data race
        - Modification order
        - Atomicity/isolation
        - Visibility
        - Ordering



Memory Model

- Locks and atomic operations communicate non-atomic writes between two threads
- Volatile is not atomic (this is not Java)
- Data races cause undefined behavior
- Some optimizations are no longer legal

Message shared memory

- Writes are explicitly communicated
  - Between pairs of threads
  - Through a lock or an atomic variable
- The mechanism is acquire and release
  - One thread releases its memory writes
    - V=32; atomic_store_explicit(&a,3, memory_order_release);
  - Another thread acquires those writes
    - i=atomic_load_explicit(&a, memory_order_acquire);

What is a memory location

- A non-bitfield primitive data object
- A sequence of adjacent bitfields
  - Not separated by a structure boundary
  - Not interrupted by the null bitfield
  - Avoid expensive atomic read-modify-write operations on bitfields

Data race condition

1. A non-atomic write to a memory location in one thread AND
2. A non-atomic read from or write to that same location in another thread AND
3. With no happens-before relations between them
   - Result in Undefined Behaviour
   - Also called “catch fire semantics”

Effect on compiler optimization

- Some rare optimizations are restricted
  - Fewer speculative writes
  - Fewer speculative reads
- Some common optimizations can be augmented
  - They may assume that loops terminate
  - Nearly always true

Atomics: To Volatile or Not Volatile

- Too much history in volatile to change its meaning
- It is not used to indicate atomicity like Java
- Volatile atomic
  - means something from the environment may also change this in addition to another thread OR
  - Real time requirements the compiler does not know about, so it further restricts compiler optimizations

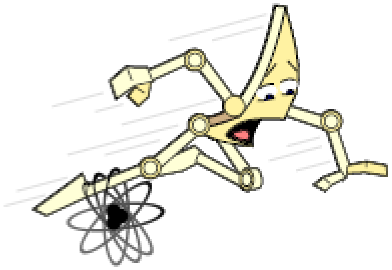
Requirements on atomics

- Static initialization
- Reasonable implementation on current hardware
- Relative novices can write working code
- Experts can performance efficient code

Consistency models

- Sequentially consistent
  - What is observed is consistent with a sequential ordering of all events in the system
    - But comes with a very heavy cost
- 2 Weaker models
  - More complex to code for some
    - But very efficient
- What we decided
  - Default is sequential consistency
  - But allow weaker semantics explicitly

Atomic Design



- Want shared variables
  - **that can be concurrently updated without introducing data race,**
  - **that are atomically updated and read**
    - **half updated states are not visible,**
- that are implemented without lock overhead whenever the hardware allows,
- that provide access to hardware atomic read-modify write (fetch-and-add, xchg, cmpxchg) instructions whenever possible.

Race Free semantics and Atomic Memory operations

- If a program has a race, it has undefined behavior
  - This is sometimes known as "catch fire" semantics
  - No compiler transformation is allowed to introduce a race
    - More restrictions on invented writes
    - Possibly fewer speculative stores **and (potentially) loads**
- There are atomic memory operations that don't cause races (or they race but are well-defined)
  - Can be used to implement locks/mutexes
  - Also useful for lock-free algorithms
- Atomic memory operations are expressed as library function calls
  - Reduces need for new language syntax

Atomic Operations and Type

- Data race: if there is no enforced ordering between two accesses to a single memory location from separate threads, one or both of those accesses is not atomic, and one or both is a write, this is a data race, and causes undefined behavior.
- These types avoid undefined behavior and provide an ordering of operations between threads

Standard Atomic Types

- atomic_flag
- atomic_bool, atomic<bool>
- Integral types:
  - atomic_char, atomic_schar, atomic_uchar, atomic_short, atomic_ushort, atomic_int, atomic_uint, atomic_long, atomic_ulong, atomic_llong, atomic_ullong atomic_char16_t, atomic_char32_t, atomic_wchar_t
  - Atomic<char>, atomic<schar>, atomic<uchar>, atomic<short>, atomic<ushort>, atomic<int>, atomic<uint>, atomic<long>, atomic<ulong>, atomic<llong>, atomic<ullong>, atomic<char16_t>, atomic<char32_t>, atomic<wchar_t>
- Typedefs like those in <cstdint>
  - atomic_int_least8_t, atomic_uint_least8_t, atomic_int_least16_t, atomic_uint_least16_t, atomic_int_least32_t, atomic_uint_least32_t, atomic_int_least64_t, atomic_uint_least64_t, atomic_int_fast8_t, atomic_uint_fast8_t, atomic_int_fast16_t, atomic_uint_fast16_t, atomic_int_fast32_t, atomic_uint_fast32_t, atomic_int_fast64_t, atomic_uint_fast64_t, atomic_intptr_t, atomic_uintptr_t, atomic_size_t, atomic_ssize_t, atomic_ptrdiff_t, atomic_intmax_t, atomic_uintmax_t

Minimal atomics

- Need 1 primitive data types that is a must, most modern hardware has instructions to implement the atomic operations
  - **for small types**
  - **and bit-wise comparison, assignment (which we require)**
  - atomic_flag type

```
static std::atomic_flag v1= ATOMIC_FLAG_INIT
If (atomic_flag_test_and_set(&v1))
    atomic_flag_clear(&v1);
```

- For other types, hardware, atomic operations may be emulated with locks.
  - Sometimes this isn.t good enough:
    - across processes, in signal/interrupt handlers.
  - is_lock_free() returns false if locks are used, and operations may block.

Basic atomics

- atomic<bool>
  - Load, store, swap, cas
- atomic<int>
  - Load, store, swap, cas
  - Fetch-and-(add, sub, and, or, xor)
- atomic<void *>
  - Load, store, swap, cas
  - Fetch-and-(add, sub)

Std::atomic<bool>

- Most basic std::atomic_bool, can be built from a non-atomic bool
- Can be constructed, initiialized, assigned from a plain bool
- assignment operator from a non-atomic bool does not return a reference to the object assigned to, but it returns a bool with the value assigned (like all other atomic types).
  - prevents code that depended on the result of the assignment to have to explicitly load the value, potentially getting a modified result from another thread.
- replace the stored value with a new one and retrieve the original value
- a plain non-modifying query of the value with an implicit conversion to plain bool
- RMW operation that stores a new value if the current value is equal to an expected value is compare_exchange_{weak/strong}();
- If we have spurious failure:

```
bool expected=false;
extern atomic_bool b; // set somewhere else
while(!b.compare_exchange_weak(expected,true) && !expected);
```

- May not be lock free, need to check per instance

Memory order strengths (C++17)

- Memory orders have the following relative strengths implied by their definitions:

relaxed --> release--------------->acq_rel--->seq_cst
\--> consume-->acquire------/

  - *x* -> *y* that means *y* is stricter/stronger than *x*

```
std::atomic<bool> b;
bool expected;
b.compare_exchange_weak(expected,true,
 memory_order_acq_rel,memory_order_acquire);
b.compare_exchange_weak(expected,true,memory_order_acq_rel);
```

Std::atomic<integral>

- this adds fetch_and, fetch_or, fetch_xor, and compound assignments like:
- +=,-=,&=,^=, pre and post increment and decrement
- missing division, multiplication and shift operations, but atomic integrals are usually used as counters or bit masks, this is not a big loss
- all semantics match fetch_add and fetch_sub for atomic_address: returns old value
- the compound assignments return new value
- ++x increments the variable and returns new value, x++ increments the variable and returns old value
- result is the value of the associated integral type

Std::atomic <> template

- std::atomic<> to create an atomic user-defined type
- Specializations for integral types derived from std::atomic_integral_type, and pointer types
- Main benefit of the template is atomic variants of user-defined types, can't be just any UDT, it must fit this criteria:
  - must have trivial copy-assignment operator: no virtual functions or virtual bases and must use the compiler-generated copt-assignment operator
  - every base class and non-static data member of UDT must also have a trivial copy-assignment operator
  - Must be bitwise equality comparable
- Only have
  - load(), store()
- Assignment and conversion to the UDT
  - exchange(), compare_exchange_weak(), compare_exchange_strong()
  - assignment from and conversion to an instance of type T

Atomic freedom

- Lock-free
  - Robust to crashes
  - Someone will make progress
  - C++14: “shall” obstruction-free
    - If there is only one unblocked thread, a lock-free execution in that thread shall complete
  - and “should” for for lock-free
    - When one or more lock-free executions run concurrently, at least one should complete.
- Wait-free
  - Operations completed in a bounded time
  - Cas vs ll/sc

Lock-free atomics

- Large atomics have no hardware support
  - Implemented with locks
- Locks and signals don’t mix
  - Test for lock-free
- Compile-time macros for basic types
  - Always lock-free
  - Never lock-free
  - May be lock-free
- RTTI for each type

Wait-free atomics

- Hard to implement without direct HW support
  - Resulting programs is usually HW-specific
  - Hard to be portable anyway
- IBM argued against it since ll/sc is not wait-free
- Few who write this seemed to cared anyway
  - No requirement for it
  - No query about it

Compiler Impact

- Memory model does not say how to make an application thread safe
  - Assumption is that source presented to compiler is thread safe
  - Undefined semantics for code with any data races
- Memory model describes legal transformations on already safe code
  - Compiler may not introduce any data races
- Memory model concerned with performance
  - Limited set of optimizations disallowed – may introduce data race
  - Allows some memory optimizations across locks
  - Use of atomics can achieve better optimization then the blunt hammer (volatile)
- Quality implementation
  - Most implementations already support a low quality implementation
  - Acquire/release operations seen as calls to opaque global functions – All shared variables may be referenced and modified

Memory Ordering Operations

```
enum memory_order {
   memory_order_relaxed,// just atomic, no onstraint
   memory_order_release,
   memory_order_acquire,
   memory_order_consume,
   memory_order_acq_rel,// both acquire and release
   memory_order_seq_cst};// sequentially consistent
```

- Every atomic operation has a default form, implicitly using `seq_cst`, and a form with an explicit order argument
- When specified, argument is expected to be just an enum constant

3 Memory Models, but 6 Memory Ordering Constraints

- Sequential Consistency
  - **Single total order for all SC ops on all variables**
  - **default**
- Acquire/Release/consume
  - **Pairwise ordering rather than total order**
  - **Independent Reads of Independent Writes don't require synchronization between CPUs**
- Relaxed Atomics
  - **Read or write data without ordering**
  - **Still obeys happens-before**
- Operations on variable have attributes, which can be explicit

Operations available on atomic types

	atomic_flag	bool/other-type	T*	integral
test_and_set, clear	Y			
is_lock_free		Y	Y	Y
load, store, exchange, compare_exchange_weak+strong		Y	Y	Y
fetch_add (+=), fetch_sub (-=), ++, --			Y	Y
fetch_or (|=), fetch_and (&=), fetch_xor (^=)				Y

Sequencing redefined for serial program

- Sequence points are … gone!
- Sequence are now defined by ordering relations
  - Sequence-before
  - Indeterminately-sequenced
- A write/write or read/write pair relations
  - That are not sequenced before
  - That are not indeterminately-sequenced
  - Results in undefined behaviour

Sequencing extended for parallel programs

- Sequenced-before
  - Provides intra-thread ordering
- Synchronizes with (Acquire and release)
  - Provide inter-thread ordering
- Happens-before relation
  - Between memory operations in different threads

Sequenced before

- If a memory update or side-effect *a is-sequenced-before* another memory operation or side-effect *b*,
  - then informally *a* must appear to be completely evaluated before *b* in the sequential execution of a single thread, e.g. all accesses and side effects of *a* must occur before those of *b*.
  - We will say that a subexpression *A* of the source program *is-sequenced-before* another subexpression *B* of the same source program to indicate that all side-effects and memory operations performed by an execution of *A* occur-before those performed by the corresponding execution of *B*, i.e. as part of the same execution of the smallest expression that includes them both.
- We propose roughly that wherever the current standard states that there is a sequence point between *A* and *B*, we instead state that *A* is-sequenced-before *B*. This will constitute the precise definition of *is-sequenced-before* on subexpressions, and hence on memory actions and side effects.

Synchronizes with

- only between operations on atomic types
- operations on a data structure (locking a mutex) might provide this relationship if the data structire contains atomic types, and the operations on that data structure perform the appropriate operations internally
- definition:
  - **a suitably-tagged atomic write operation on a variable x synchronizes-with a suitably-tagged atomic read operation on x that reads the value stored**
    by (a) that write, (b) a subsequent atomic write operation on x by the same thread that
    performed the initial write, or (c) an atomic read-modify-write operation on x (such as
    fetch_add() or compare_exchange_weak()) by any thread, that read the value written.
- Store-release synchronizes-with a load-acquire

Happens before

- It specifies which operations see the effects of which other operations.
- An evaluation A happens before an evaluation B if:
  - **A is sequenced before B, or**
  - **A synchronizes with B, or**
  - **for some evaluation X, A happens before X and X happens before B.**

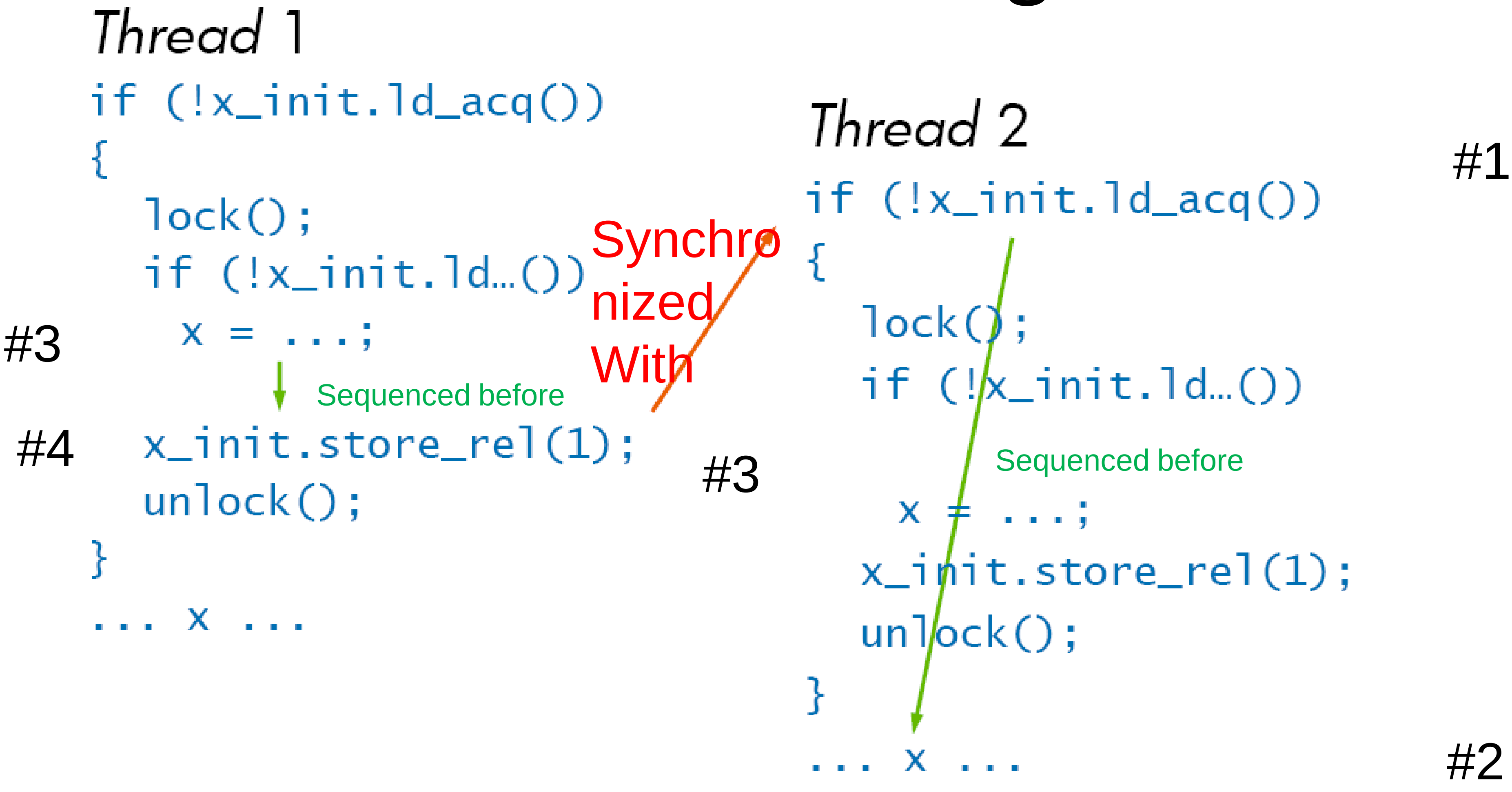
Double-Checked Locking Pattern that works (from Fedor Pikus)

- For all calls except the first one, the_gizmo is not NULL and atomic check-assignment is not needed

```
static Atomic<Gizmo*> the_gizmo = NULL;
if (the_gizmo.AcquireLoad() == NULL) {
  Lock(&mutex);
  if (the_gizmo.NonAtomicLoad() == NULL) {
    the_gizmo.ReleaseStore(new Gizmo);
  }
 Unlock(&mutex);
}
```

- Lock-free programming is 90% about memory visibility and order of memory accesses
- Atomic instructions are the easy part

Happens-before using double checked locking



Thread 1

```
if (!x_init.ld_acq())
{
   lock();
   if (!x_init.ld…())
     x = …;
   x_init.store_rel(1);
   unlock();
}
… x …
```

#3

Sequenced before

#4

Synchronized With

#3

Thread 2

```
if (!x_init.ld_acq())
{
   lock();
   if (!x_init.ld…())
     x = …;
   x_init.store_rel(1);
   unlock();
}
… x …
```

Sequenced before

#1

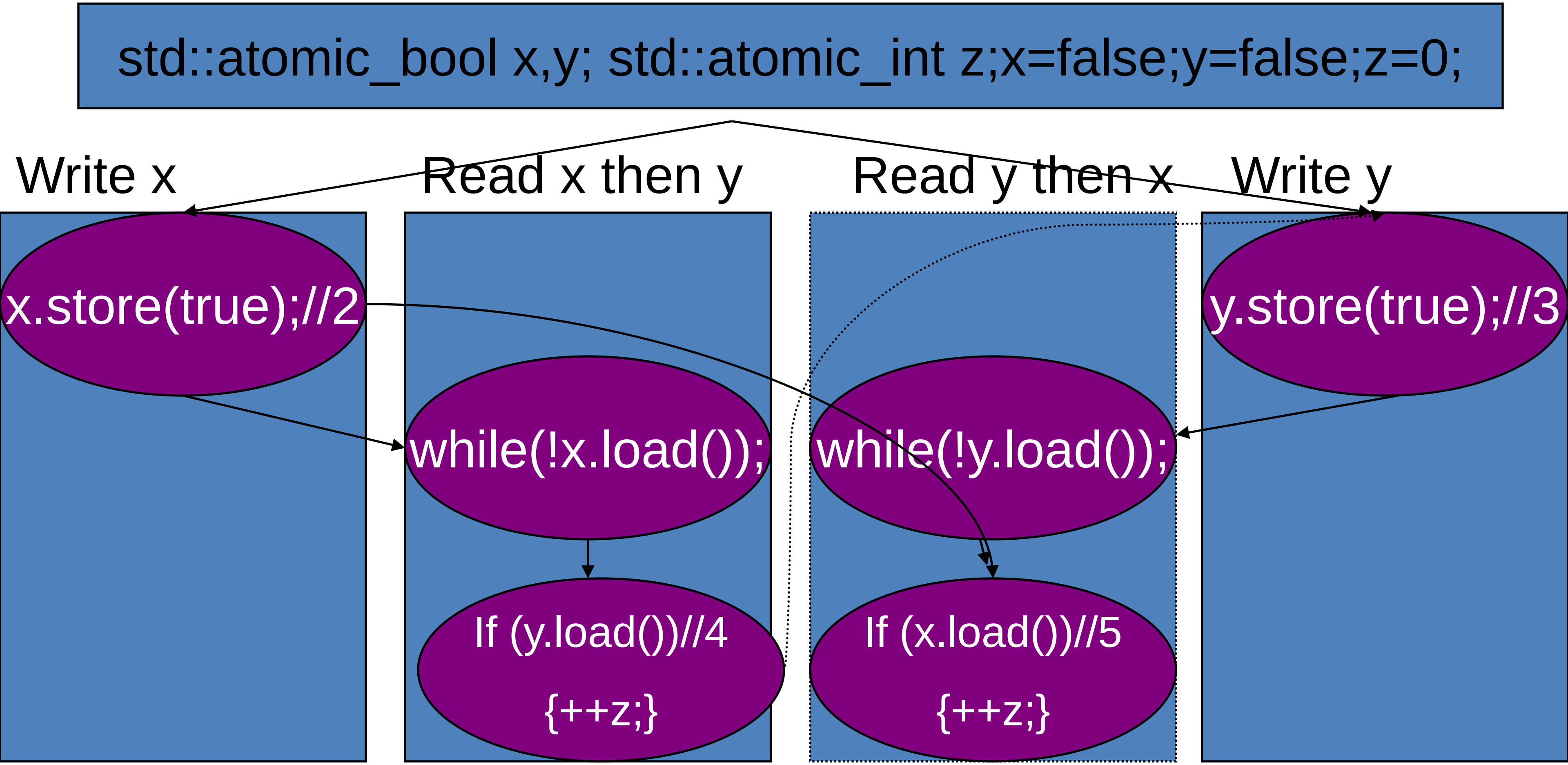
#2

Sequential Consistency implies a total ordering

```
std::atomic_bool x,y;
std::atomic_int z;
void write_x()
{
        x.store(true,std::memory_order_seq_cst);
        //2
}
void write_y()
{
        y.store(true,std::memory_order_seq_cst);
        //3
}
void read_x_then_y()
{
     while(!x.load(std::memory_order_seq_cst));
     if(y.load(std::memory_order_seq_cst)) //4
          ++z;
}
void read_y_then_x()
{
     while(!y.load(std::memory_order_seq_cst));
     if(x.load(std::memory_order_seq_cst)) //5
          ++z;
}
```

```
int main()
{
   x=false;
   y=false;
   z=0;
   std::thread a(write_x);
   std::thread b(write_y);
   std::thread c(read_x_then_y);
   std::thread d(read_y_then_x);
   a.join();
   b.join();
   c.join();
   d.join();
   assert(z.load()!=0); //1
}
```

SC and happens-before



```
std::atomic_bool x,y; std::atomic_int z;x=false;y=false;z=0;
```

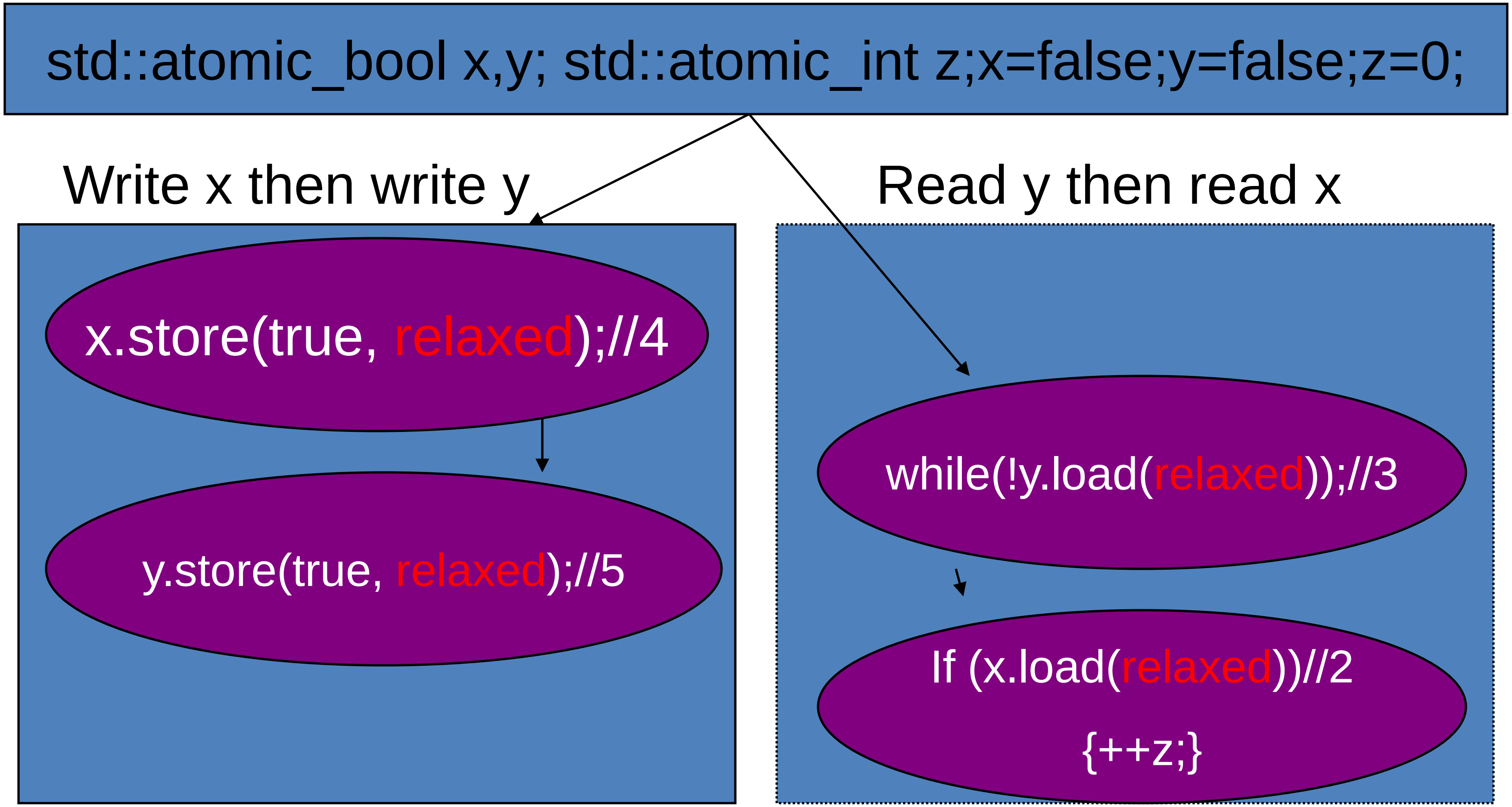
```
assert(z.load()!=0); //1
```

Relaxed operations have little ordering requirement

```
std::atomic_bool x,y;
std::atomic_int z;
void write_x_then_y()
{
   x.store(true,std::memory_order_relaxed); //4
   y.store(true,std::memory_order_relaxed); //5
}
void read_y_then_x()
{
   while(!y.load(std::memory_order_relaxed)); //3
   if(x.load(std::memory_order_relaxed)) //2
      ++z;
}
```

```
int main()
{
   x=false;
   y=false;
   z=0;
   std::thread a(write_x_then_y);
   std::thread b(read_y_then_x);
   a.join();
   b.join();
   assert(z.load()!=0); //1
}
```

Relaxed and happens-before



std::atomic_bool x,y; std::atomic_int z;x=false;y=false;z=0;

Write x then write y

x.store(true, relaxed);//4

y.store(true, relaxed);//5

Read y then read x

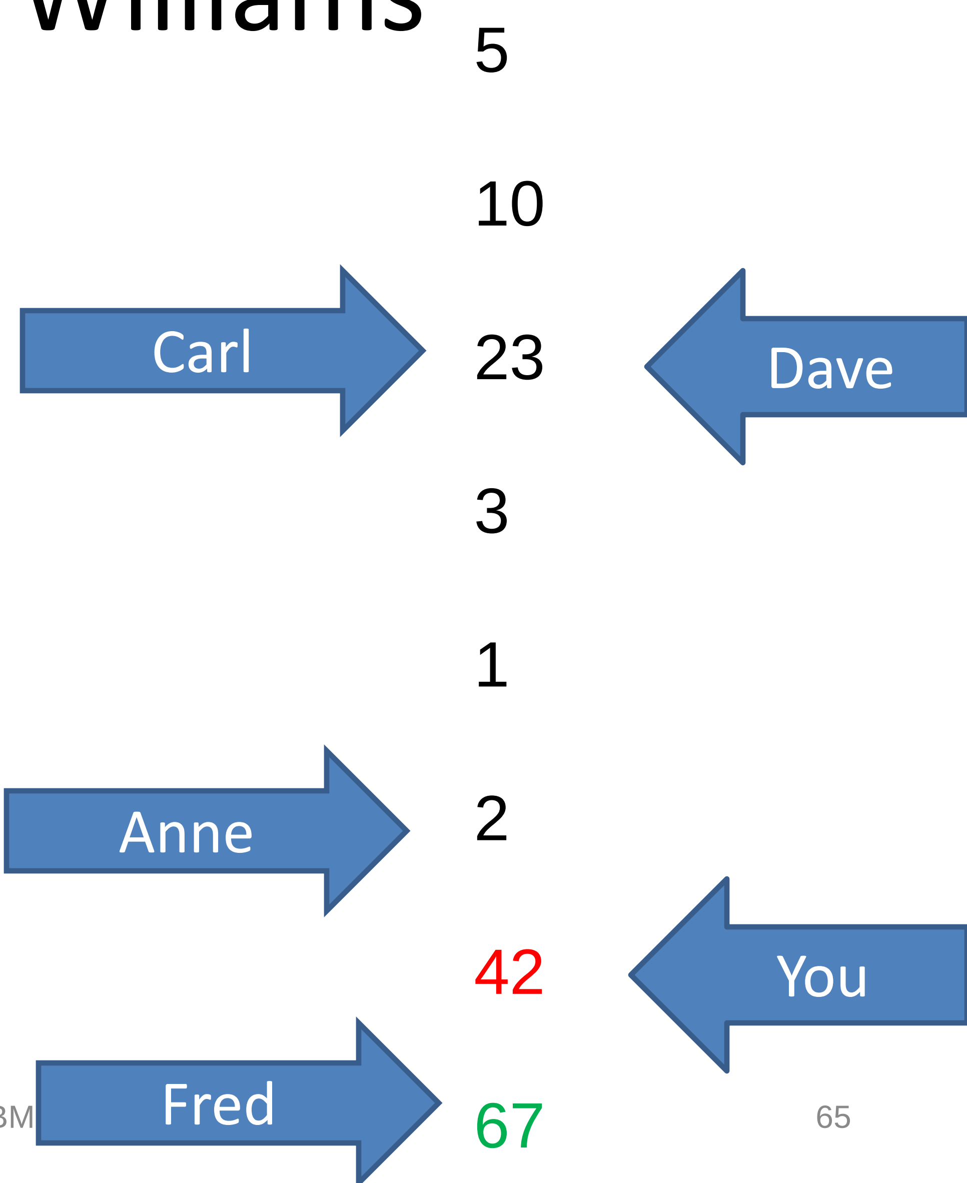
while(!y.load(relaxed));//3

If (x.load(relaxed))//2

{++z;}

assert(z.load()!=0); //1

Taking orders over the phone from Anthony Williams



5

10

Carl 23 Dave

3

1

Anne 2

42 You

Fred 67

Food for thought and Q/A

- This is the chance to get a copy before you have to pay for it:
  - **https://ctweb.torolab.ibm.com/wiki/index.php/Atomics**
  - **http://www.hpl.hp.com/personal/Hans_Boehm/c++mm**
  - C++ : http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3291.pdf
  - C++ (last free version): http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
  - C: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
- Participate and feedback to Compiler
  - What features/libraries interest you or your customers?
  - What problem/annoyance you would like the Std to resolve?
  - Is Special Math important to you?
  - Do you expect 0x features to be used quickly by your customers?
- Talk to me at my blog:
  - http://www.ibm.com/software/rational/cafe/blogs/cpp-standard

My blogs and email address

- http://wongmichael.com
- ISOCPP.org Director, VP
  http://isocpp.org/wiki/faq/wg21#michael-wong
  OpenMP CEO: http://openmp.org/wp/about-openmp/
  My Blogs: http://ibm.co/pCvPHR
  C++11 status: http://tinyurl.com/43y8xgf
  Boost test results
  http://www.ibm.com/support/docview.wss?rs=2239&context=SSJT9L&uid=swg27006911
  C/C++ Compilers Feature Request Page
  http://www.ibm.com/developerworks/rfe/?PROD_ID=700
  Chair of WG21 SG5 Transactional MemoryM:
  https://groups.google.com/a/isocpp.org/forum/?hl=en&fromgroups#!forum/tm

Acknowledgement

- Some slides are based on committee presentations by various committee members, their proposals, and private communication

My blogs and email address

- **ISOCPP.org Director, VP http://isocpp.org/wiki/faq/wg21#michael-wong OpenMP CEO: http://openmp.org/wp/about-openmp/ My Blogs: http://ibm.co/pCvPHR C++11 status: http://tinyurl.com/43y8xgf Boost test results http://www.ibm.com/support/docview.wss?rs=2239&context=SSJT9L&uid=swg27006911 C/C++ Compilers Feature Request Page http://www.ibm.com/developerworks/rfe/?PROD_ID=700 Chair of WG21 SG5 Transactional MemoryM: https://groups.google.com/a/isocpp.org/forum/?hl=en&fromgroups#!forum/tm**