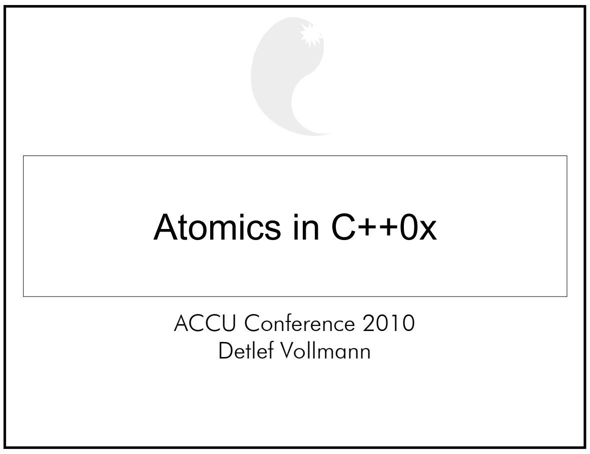
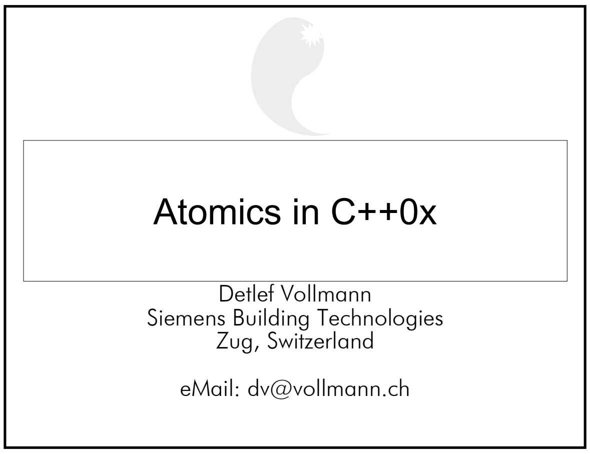
Atomics in C++0x



Atomics in C++0x

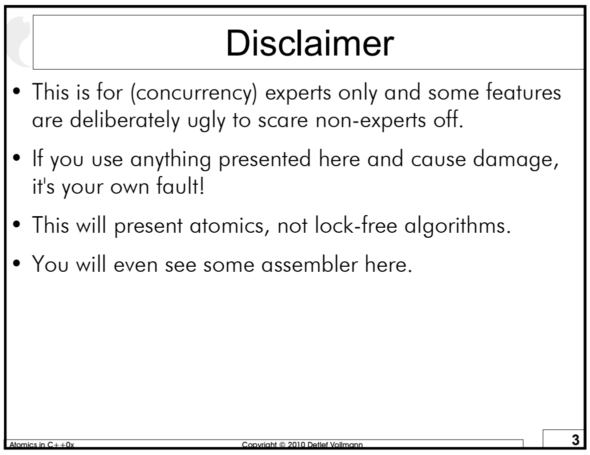
ACCU Conference 2010
Detlef Vollmann



Atomics in C++0x

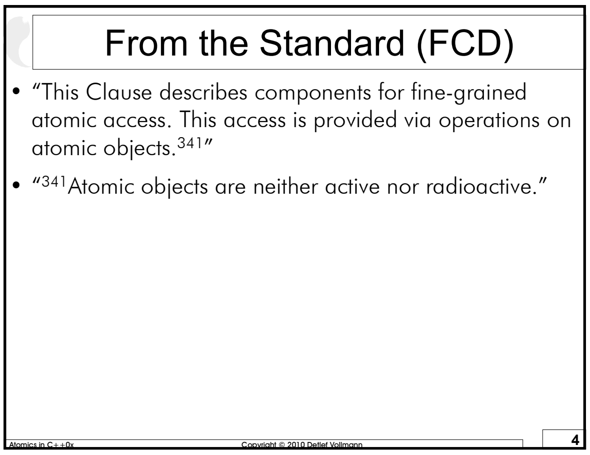
Detlef Vollmann
Siemens Building Technologies
Zug, Switzerland

eMail: dv@vollmann.ch



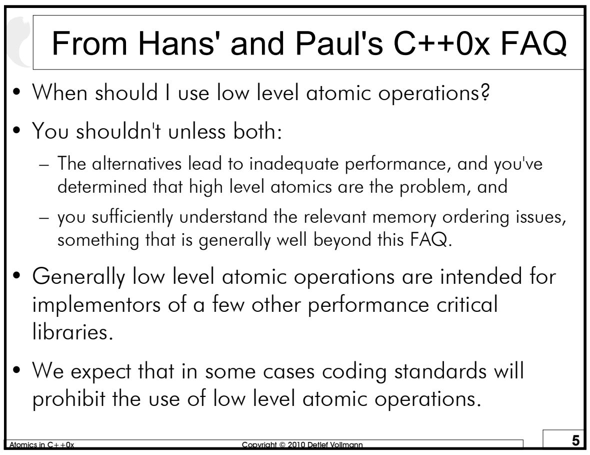
Disclaimer

- This is for (concurrency) experts only and some features are deliberately ugly to scare non-experts off.
- If you use anything presented here and cause damage, it's your own fault!
- This will present atomics, not lock-free algorithms.
- You will even see some assembler here.



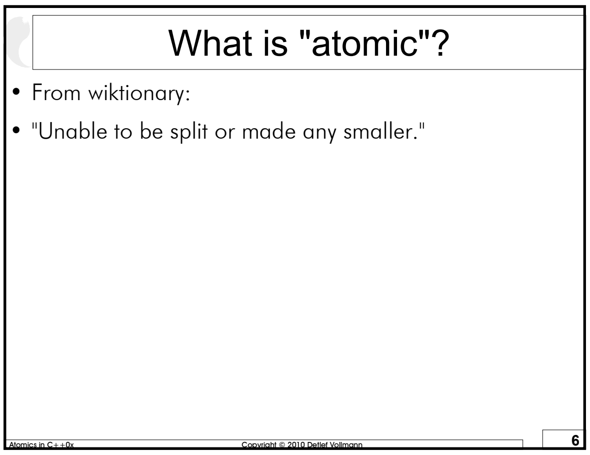
From the Standard (FCD)

- “This Clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.[341]”
- “[341]Atomic objects are neither active nor radioactive.”



From Hans' and Paul's C++0x FAQ

- When should I use low level atomic operations?
- You shouldn't unless both:
  - The alternatives lead to inadequate performance, and you've determined that high level atomics are the problem, and
  - you sufficiently understand the relevant memory ordering issues, something that is generally well beyond this FAQ.
- Generally low level atomic operations are intended for implementors of a few other performance critical libraries.
- We expect that in some cases coding standards will prohibit the use of low level atomic operations.



What is "atomic"?

- From wiktionary:
- "Unable to be split or made any smaller."

Atomic Increment

- Which one is atomic:

```
      addi    #1,%r0
```

- or

```
1:    ldarx   r1,0,%r0
      addi    r1,r1,1
      stdcx   r1,0,%r0
      bne     1
```

- or

```
void atomic_inc_32(volatile uint32_t *addr) {
  uint32_t old, new;
  do {
     old = *addr;
     new = old + 1;
  } while (atomic_cas_32(addr, old, new) != old);
}
```

What is atomic?

- From wiktionary:
- "Said of an operation that is guaranteed to either complete fully, or not at all."
- So an atomic operation is a transaction.

Singleton

```
Precious* Precious::Instance() {
  lock_guard<mutex> lock(mtx);   // lock
  if (!pInstance) {
     pInstance = new Precious;
  }
  return pInstance;
}                                // unlock
```

- Each access aquires the lock.
- This costs something.
- Some "experts" believe they can do better...

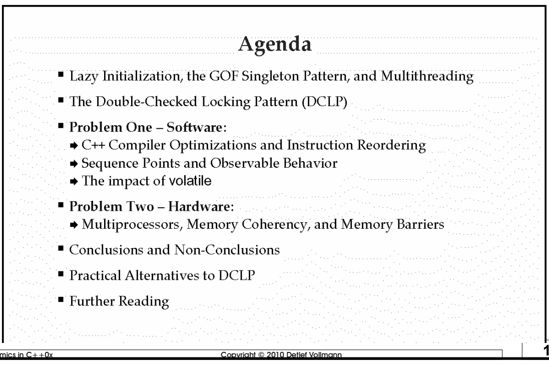
Double-Checked Locking (DCL)

```
class Precious {
  volatile Precious *pInstance;
public:
  Precious *Instance() {
    if (!pInstance) {
      lock_guard<mutex> lock(mtx);   // lock
      if (!pInstance) {
        pInstance = new Precious;
      }                              // unlock
    }
    return pInstance;
  }
};
```

Double-Checked

- It's clever and ... unsafe.
- See the "Double-Checked Locking is Broken" Declaration, available at http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

Scott Meyer's Analysis of Double-Checked Locking (DCL)



Agenda

- Lazy Initialization, the GOF Singleton Pattern, and Multithreading
- The Double-Checked Locking Pattern (DCLP)
- **Problem One – Software:**
  - C++ Compiler Optimizations and Instruction Reordering
  - Sequence Points and Observable Behavior
  - The impact of volatile
- **Problem Two – Hardware:**
  - Multiprocessors, Memory Coherency, and Memory Barriers
- Conclusions and Non-Conclusions
- Practical Alternatives to DCLP
- Further Reading

Analysis of Double-Checked Locking (DCL)

- Scott was essentially right.
- Third problem: Hardware Instruction Re-Ordering
- Atomics are here to solve them all!
- Really?

Compiler Optimization

- From ISO/IEC JTC1 SC22 WG21 N2427, "C++ Atomic Types and Operations" by Hans Boehm and Lawrence Crowl:
- "We propose to add atomic types and operations purely as a library API. In practice, for C++, **this API would have to be implemented largely with either compiler intrinsics or assembly code.** (As such, this proposal should be implemented by compiler vendors, not library vendors, much as the exception facilities are implemented by compiler vendors.) For C, a compiler implementation is required for the type-generic macros."
- Atomics are here to inhibit unwanted compiler optimizations.

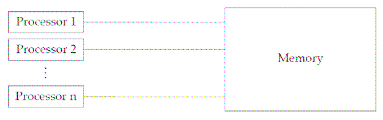
Hardware Instruction Re-Ordering

- From a PopwerPC manual (MPC885 PowerQUICC(TM) Family Reference Manual):
- "The isync instruction is context synchronizing, which **guarantees that all of the effects of previous instructions are in place and any instructions in the instruction queue are flushed** (which means all instructions that were in the instruction queue need to be refetched)."
- "The sync instruction delays execution of subsequent instructions until previous instructions have completed to the point that they can no longer cause an exception and until all previous memory accesses are performed globally; the sync operation is not broadcast onto the MPC885 bus interface. **Additionally, all load and store cache/bus activities initiated by prior instructions are completed.**"

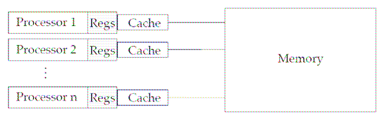
Memory Coherency

Multiprocessors and Memory Coherency

Conceptually, multiprocessor systems often look like this:

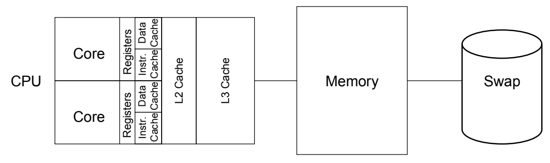


But this is more accurate:



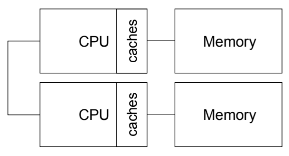
Registers and cache are examples of processor-specific *local memory*.

Multi-Core Memory Hierarchy

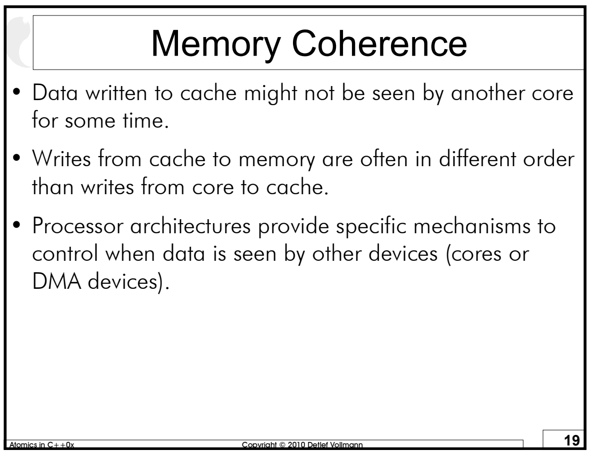


- Multiple complete CPUs inside one package
  - single or multiple die
- Caches might be shared or separate
- Communication between cores (caches) might be internal or external

NUMA

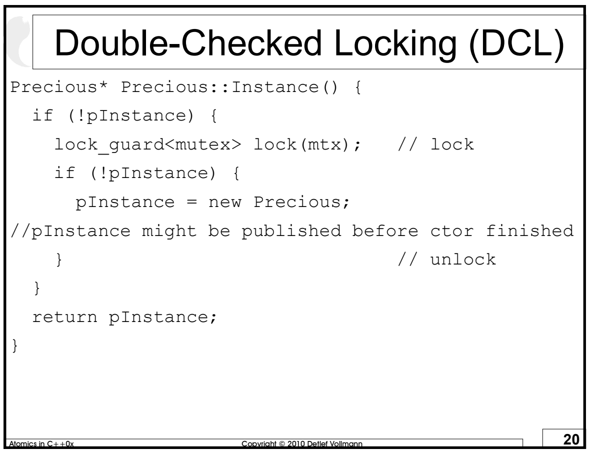


- Non-Uniform Memory Access
- Single address space
- Access to remote memory via other CPU
- ccNUMA: cache coherent NUMA



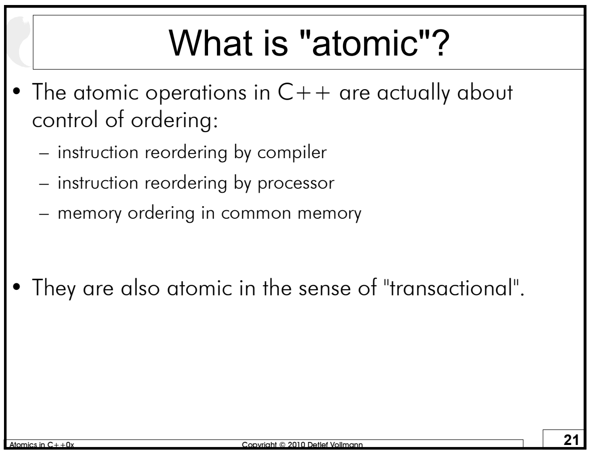
Memory Coherence

- Data written to cache might not be seen by another core for some time.
- Writes from cache to memory are often in different order than writes from core to cache.
- Processor architectures provide specific mechanisms to control when data is seen by other devices (cores or DMA devices).



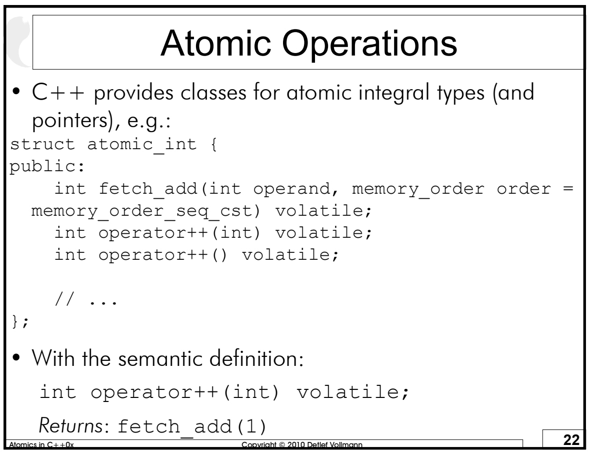
Double-Checked Locking (DCL)

```
Precious* Precious::Instance() {
  if (!pInstance) {
    lock_guard<mutex> lock(mtx);   // lock
    if (!pInstance) {
      pInstance = new Precious;
//pInstance might be published before ctor finished
    }                              // unlock
  }
  return pInstance;
}
```



What is "atomic"?

- The atomic operations in C++ are actually about control of ordering:
  - instruction reordering by compiler
  - instruction reordering by processor
  - memory ordering in common memory
- They are also atomic in the sense of "transactional".



Atomic Operations

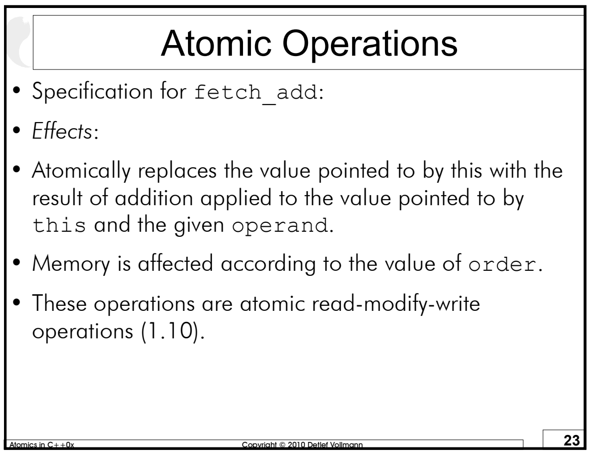
- C++ provides classes for atomic integral types (and pointers), e.g.:

```
struct atomic_int {
public:
    int fetch_add(int operand, memory_order order =
  memory_order_seq_cst) volatile;
    int operator++(int) volatile;
    int operator++() volatile;

    // ...
};
```

- With the semantic definition:

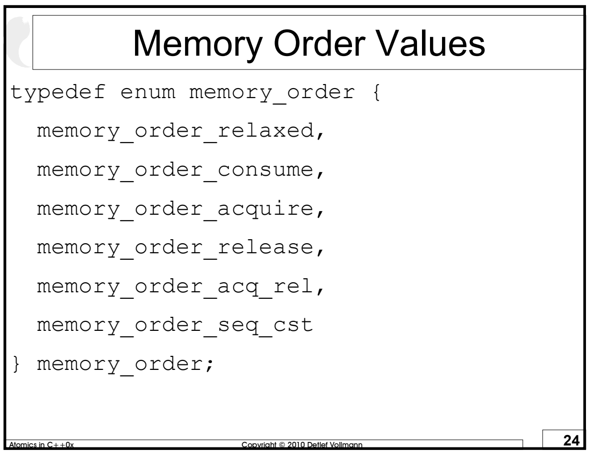
  `int operator++(int) volatile;`

  Returns: `fetch_add(1)`



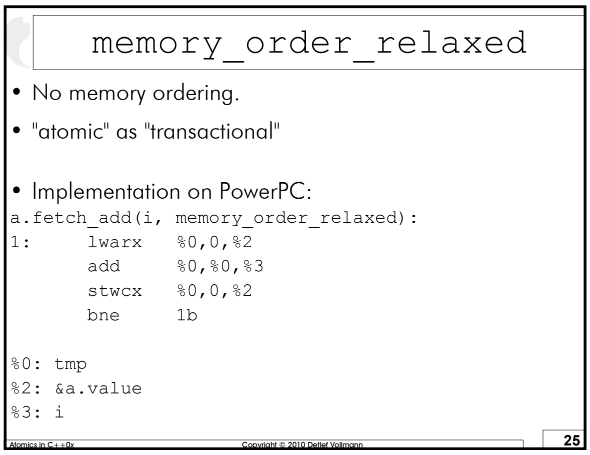
Atomic Operations

- Specification for `fetch_add`:
- *Effects*:
- Atomically replaces the value pointed to by this with the result of addition applied to the value pointed to by `this` and the given `operand`.
- Memory is affected according to the value of `order`.
- These operations are atomic read-modify-write operations (1.10).



Memory Order Values

```
typedef enum memory_order {
  memory_order_relaxed,
  memory_order_consume,
  memory_order_acquire,
  memory_order_release,
  memory_order_acq_rel,
  memory_order_seq_cst
} memory_order;
```

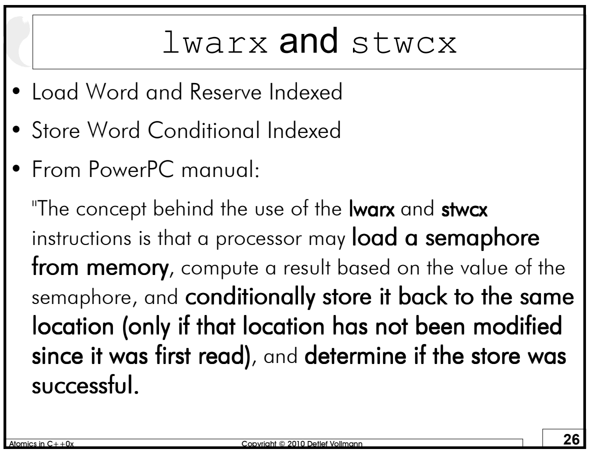


memory_order_relaxed

- No memory ordering.
- "atomic" as "transactional"

- Implementation on PowerPC:

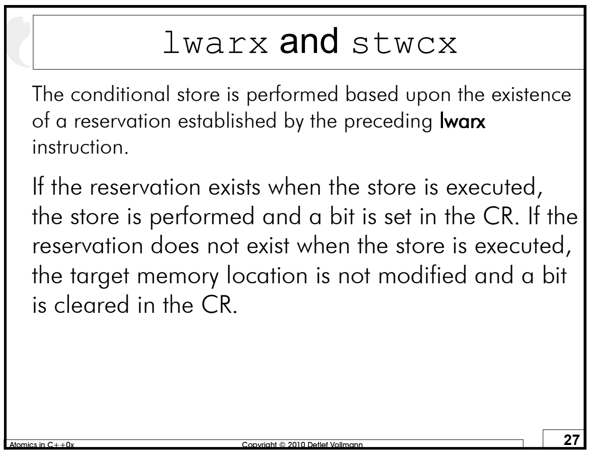
```
a.fetch_add(i, memory_order_relaxed):
1:      lwarx   %0,0,%2
        add     %0,%0,%3
        stwcx   %0,0,%2
        bne     1b

%0: tmp
%2: &a.value
%3: i
```



lwarx and stwcx

- Load Word and Reserve Indexed
- Store Word Conditional Indexed
- From PowerPC manual:

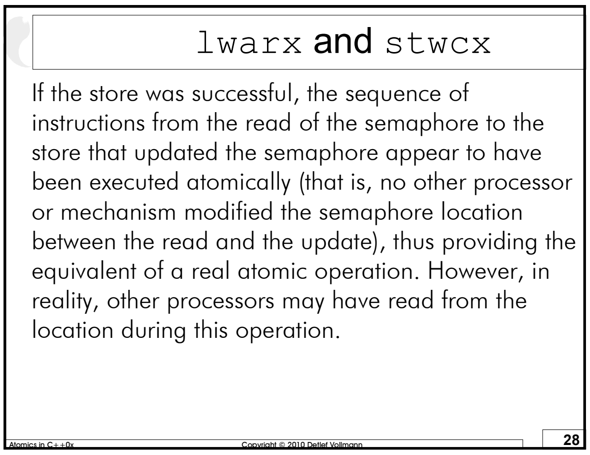
"The concept behind the use of the **lwarx** and **stwcx** instructions is that a processor may **load a semaphore from memory**, compute a result based on the value of the semaphore, and **conditionally store it back to the same location (only if that location has not been modified since it was first read)**, and **determine if the store was successful.**



lwarx and stwcx

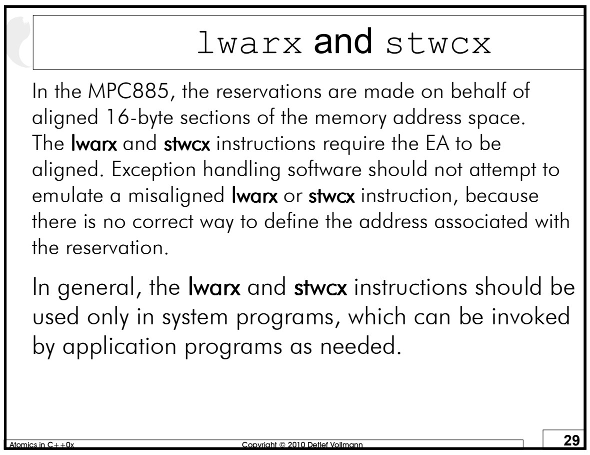
The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx** instruction.

If the reservation exists when the store is executed, the store is performed and a bit is set in the CR. If the reservation does not exist when the store is executed, the target memory location is not modified and a bit is cleared in the CR.



lwarx and stwcx

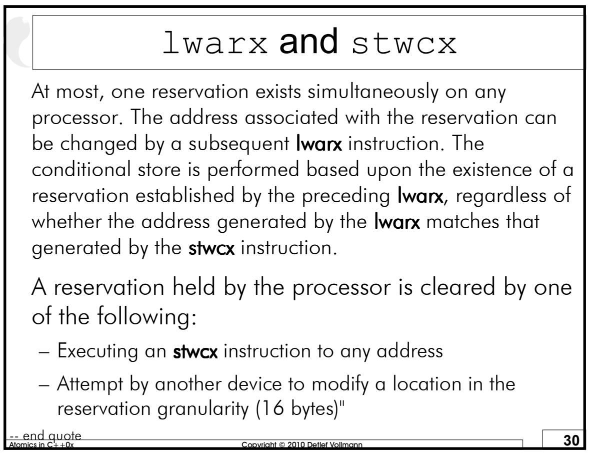
If the store was successful, the sequence of instructions from the read of the semaphore to the store that updated the semaphore appear to have been executed atomically (that is, no other processor or mechanism modified the semaphore location between the read and the update), thus providing the equivalent of a real atomic operation. However, in reality, other processors may have read from the location during this operation.



lwarx and stwcx

In the MPC885, the reservations are made on behalf of aligned 16-byte sections of the memory address space. The **lwarx** and **stwcx** instructions require the EA to be aligned. Exception handling software should not attempt to emulate a misaligned **lwarx** or **stwcx** instruction, because there is no correct way to define the address associated with the reservation.

In general, the **lwarx** and **stwcx** instructions should be used only in system programs, which can be invoked by application programs as needed.



lwarx and stwcx

At most, one reservation exists simultaneously on any processor. The address associated with the reservation can be changed by a subsequent **lwarx** instruction. The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx**, regardless of whether the address generated by the **lwarx** matches that generated by the **stwcx** instruction.

A reservation held by the processor is cleared by one of the following:

- Executing an **stwcx** instruction to any address
- Attempt by another device to modify a location in the reservation granularity (16 bytes)"

-- end quote

memory_order_relaxed

```
// Initialization:
atomic<int> x(0), y(0);
int r1, r2;

// Thread 1:
r1 = y.load(memory_order_relaxed);    (2)
x.store(r1, memory_order_relaxed);    (3)

// Thread 2:
r2 = x.load(memory_order_relaxed);    (4)
y.store(42, memory_order_relaxed);    (1)
```

Result: `r1 = r2 = 42`.

memory_order_consume

- "don't move any subsequent 'dependent' memory load before this operation"
- Some implementations implement this by just doing a `memory_order_aquire`.

memory_order_acquire

- also RMB: read memory barrier
- "don't move any subsequent load before this barrier"
- or
- "invalidate any cache locations that are read after this"
- This should be done if you enter some synchronized section.

memory_order_release

- also WMB: write memory barrier
- "don't move any previous store after this barrier"
- or
- "publish everything that's done until now"
- This should be done on leaving some synchronized section.

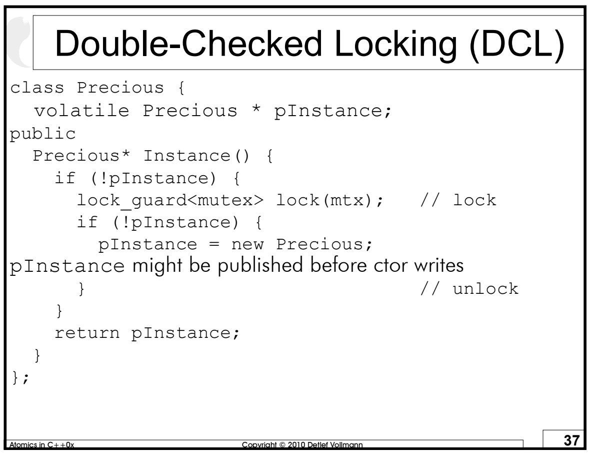
memory_order_acq_rel

- Full memory barrier: both, read and write memory barrier.
- "Don't move any memory access across this barrier."

memory_order_seq_cst

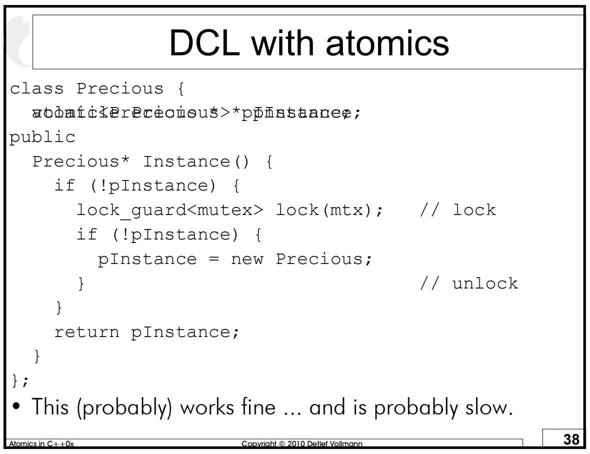
- Sequential consistency:
- "Don't move any instructions."
- This is about hardware instruction reordering.
  - and full memory ordering.
- This is pretty safe.
  - ... and it's the default.
- And as Bartosz Milewski says:
  "Any time you deviate from sequential consistency, you increase the complexity of the problem by orders of magnitude."

http://bartoszmilewski.wordpress.com/2008/12/01/c-atomics-and-memory-ordering/



Double-Checked Locking (DCL)

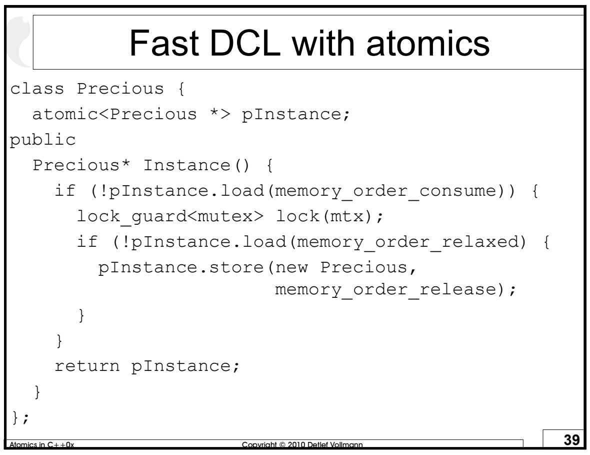
```
class Precious {
  volatile Precious * pInstance;
public
  Precious* Instance() {
    if (!pInstance) {
      lock_guard<mutex> lock(mtx);   // lock
      if (!pInstance) {
        pInstance = new Precious;
```
`pInstance` might be published before ctor writes
```
      }                              // unlock
    }
    return pInstance;
  }
};
```



DCL with atomics

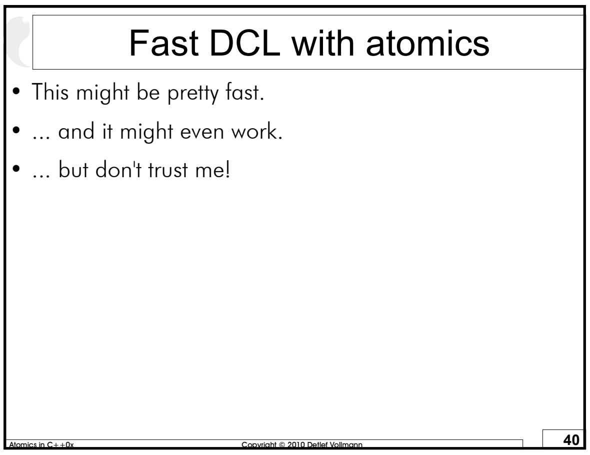
```
class Precious {
  atomic<Precious *> pInstance;
  volatile Precious * pInstance;
public
  Precious* Instance() {
    if (!pInstance) {
      lock_guard<mutex> lock(mtx);   // lock
      if (!pInstance) {
        pInstance = new Precious;
      }                              // unlock
    }
    return pInstance;
  }
};
```

- This (probably) works fine ... and is probably slow.



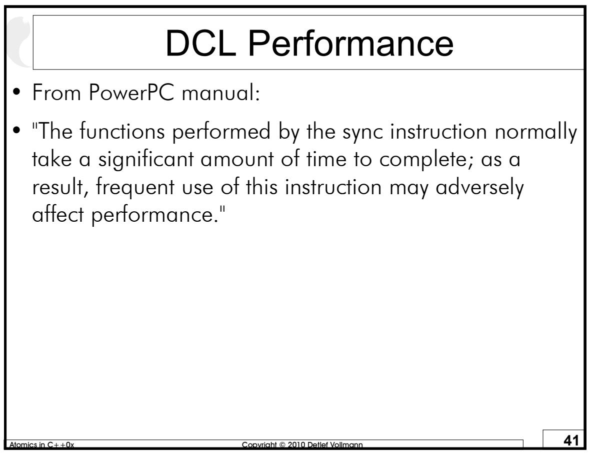
Fast DCL with atomics

```
class Precious {
  atomic<Precious *> pInstance;
public
  Precious* Instance() {
    if (!pInstance.load(memory_order_consume)) {
      lock_guard<mutex> lock(mtx);
      if (!pInstance.load(memory_order_relaxed) {
        pInstance.store(new Precious,
                        memory_order_release);
      }
    }
    return pInstance;
  }
};
```



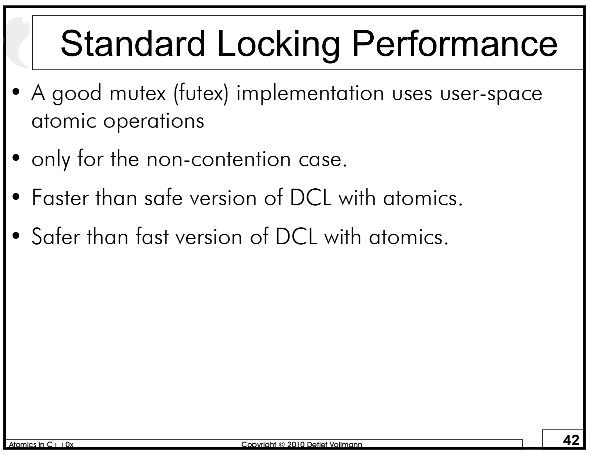
Fast DCL with atomics

- This might be pretty fast.
- ... and it might even work.
- ... but don't trust me!



DCL Performance

- From PowerPC manual:
- "The functions performed by the sync instruction normally take a significant amount of time to complete; as a result, frequent use of this instruction may adversely affect performance."



Standard Locking Performance

- A good mutex (futex) implementation uses user-space atomic operations
- only for the non-contention case.
- Faster than safe version of DCL with atomics.
- Safer than fast version of DCL with atomics.

When To Use atomics?

- When you're an expert.
- When you need lock-free behaviour.
- E.g. interrupt service routines.

Atomic Operations

- `struct atomic_flag`:
- `bool test_and_set(memory_order)`
  - sets the flag, returns old value
- `void clear(memory_order)`
  - clears the flag
- no `load()`!

`atomic<int>` Operations

- Actually for all integer types.
- `atomic(int)`: non atomic
- `store`/`load`
- `operator int()`: atomic conversion (load)
- `operator=()`: just a seq_cst store
- `exchange(new_value, order)`: returns old value
- `fetch_add`, `fetch_sub`, `fetch_and`, `fetch_or`, `fetch_xor`: returns old value
- `+=`, `-=`, `&=`, `|=`, `^=`: return new value
- `++`, `--`

Compare And Swap

```
• bool compare_exchange_weak(
      int &expected,
      int desired,
      memory_order success,
      memory_order failure);
• bool compare_exchange_strong(...)

  if (*this == expected) {
    store(desired, success);
    return true;
  } else {
    expected = load(failure)
    return false;
  }
```

- The weak version may fail spuriously.

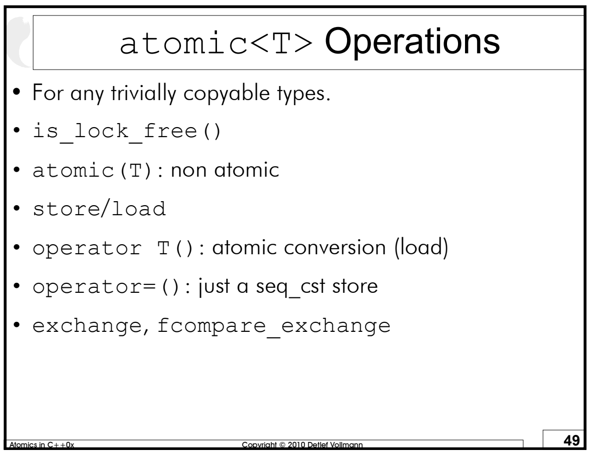
`atomic<T *>` Operations

- like `atomic<int>`
- but no `fetch_and/or/xor`
- no operators `*` or `[]`

ABA Problem

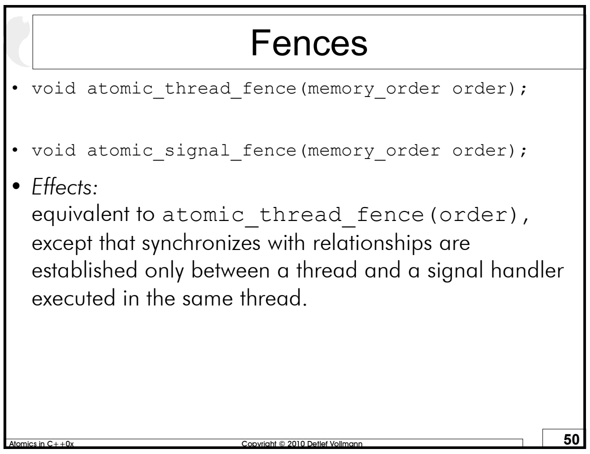
- CAS principle:
  - load old value
  - prepare new data based on old value
  - run `compare_exchange` with old value as expected

  ⇒ if we still find the old value, nothing has changed
- Wrong!
- Another thread might have changed it twice, to another value and back.
- This might or might not be a problem.
- Workaround: use a two-word `compare_exchange` with an additional counter



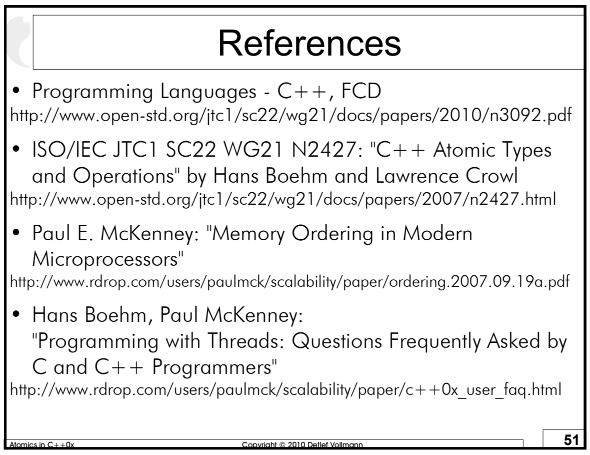
`atomic<T>` Operations

- For any trivially copyable types.
- `is_lock_free()`
- `atomic(T)`: non atomic
- `store/load`
- `operator T()`: atomic conversion (load)
- `operator=()`: just a seq_cst store
- `exchange`, `fcompare_exchange`



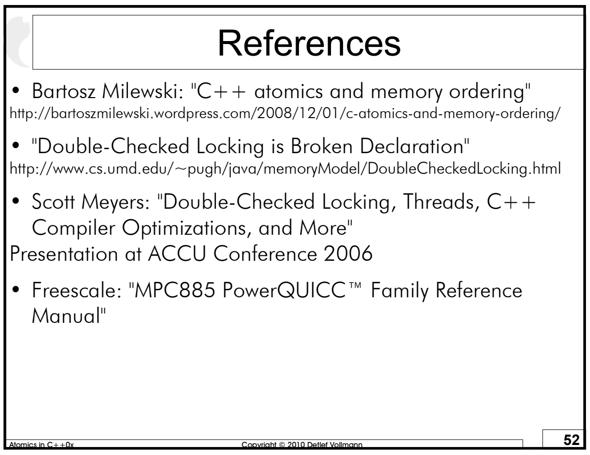
Fences

- `void atomic_thread_fence(memory_order order);`
- `void atomic_signal_fence(memory_order order);`
- *Effects:*
  equivalent to `atomic_thread_fence(order)`, except that synchronizes with relationships are established only between a thread and a signal handler executed in the same thread.



References

- Programming Languages - C++, FCD
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf
- ISO/IEC JTC1 SC22 WG21 N2427: "C++ Atomic Types and Operations" by Hans Boehm and Lawrence Crowl
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html
- Paul E. McKenney: "Memory Ordering in Modern Microprocessors"
http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.19a.pdf
- Hans Boehm, Paul McKenney:
  "Programming with Threads: Questions Frequently Asked by C and C++ Programmers"
http://www.rdrop.com/users/paulmck/scalability/paper/c++0x_user_faq.html



References

- Bartosz Milewski: "C++ atomics and memory ordering"
http://bartoszmilewski.wordpress.com/2008/12/01/c-atomics-and-memory-ordering/
- "Double-Checked Locking is Broken Declaration"
http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html
- Scott Meyers: "Double-Checked Locking, Threads, C++ Compiler Optimizations, and More"
Presentation at ACCU Conference 2006
- Freescale: "MPC885 PowerQUICC™ Family Reference Manual"