# A Systematic View of Atomics

Adam Cavendish

2017.12.16

# Outline

Concurrency, Shared Memory Model, Threads

Basic Intro to Modern C++ Atomics

Optimization, Races, Perspectives from Hardware and Software

General Acquire and Release Concepts

x86_64 Memory Ordering Model

C++ Memory Ordering

Atomic Thread Fence

C++ Atomics Pitfalls

Some Correct Use Examples

# Concurrency, Shared Memory Model, Threads

# Concurrency Is Hard

**A simple guide to concurrency**

1. Stop sharing data
2. Do use locks
3. Profiling
4. Profiling
5. Profiling
6. Improve your algorithm
7. Goto No. 1

…

∞. Try Lock-free

# Data Race

1. At least two concurrent accesses, and
2. One is unsynchronized, or one is write

# Multithreading

Naturally Shares Everything / Use Same Address Space

# Basic Intro to Modern C++ Atomics

# Related Information

1. C++ Reference
   - `http://en.cppreference.com/w/cpp/atomic`
2. C++ Concurrency In Action (Anthony Williams)
3. Intel Software Developer Manuals
4. HSA Programmer's Reference Manual

# Related Information, Cont'd

4. The Perf Book (Paul E. McKenney)
5. The Art of Multiprocessor Programming (Maurice Herlihy & Nir Shavit)
6. Java Memory Model Cookbook JSR-133

# Misunderstandings About Lock-free Programming

## Rumors about Lock-free Programming

# Misunderstandings About Lock-free Programming

**Rumors about Lock-free Programming**

**Bindly Believe in Atomic Instructions**

1. Always apply serial programming experience to concurrent problems
2. Lack of Computer Archtechture Knowledge

# Misunderstandings About Lock-free Programming, Cont'd

## Poor Man's Atomics

```
std::atomic<int> atomic_sum {0};
std::array<int, N> arr;
void work() {
  for (size_t i = 0; i < arr.size(); ++i)
    atomic_sum += arr[i];
}
```

## General Mutex Version

```
int sum = 0;
std::mutex sum_mutex;
void work() {
  int local_sum = 0;
  for (size_t i = 0; i < arr.size(); ++i)
    local_sum += arr[i];
  std::lock_guard<std::mutex> lock(sum_mutex);
  sum += local_sum;
}
```

# Misunderstandings About Lock-free Programming, Cont'd

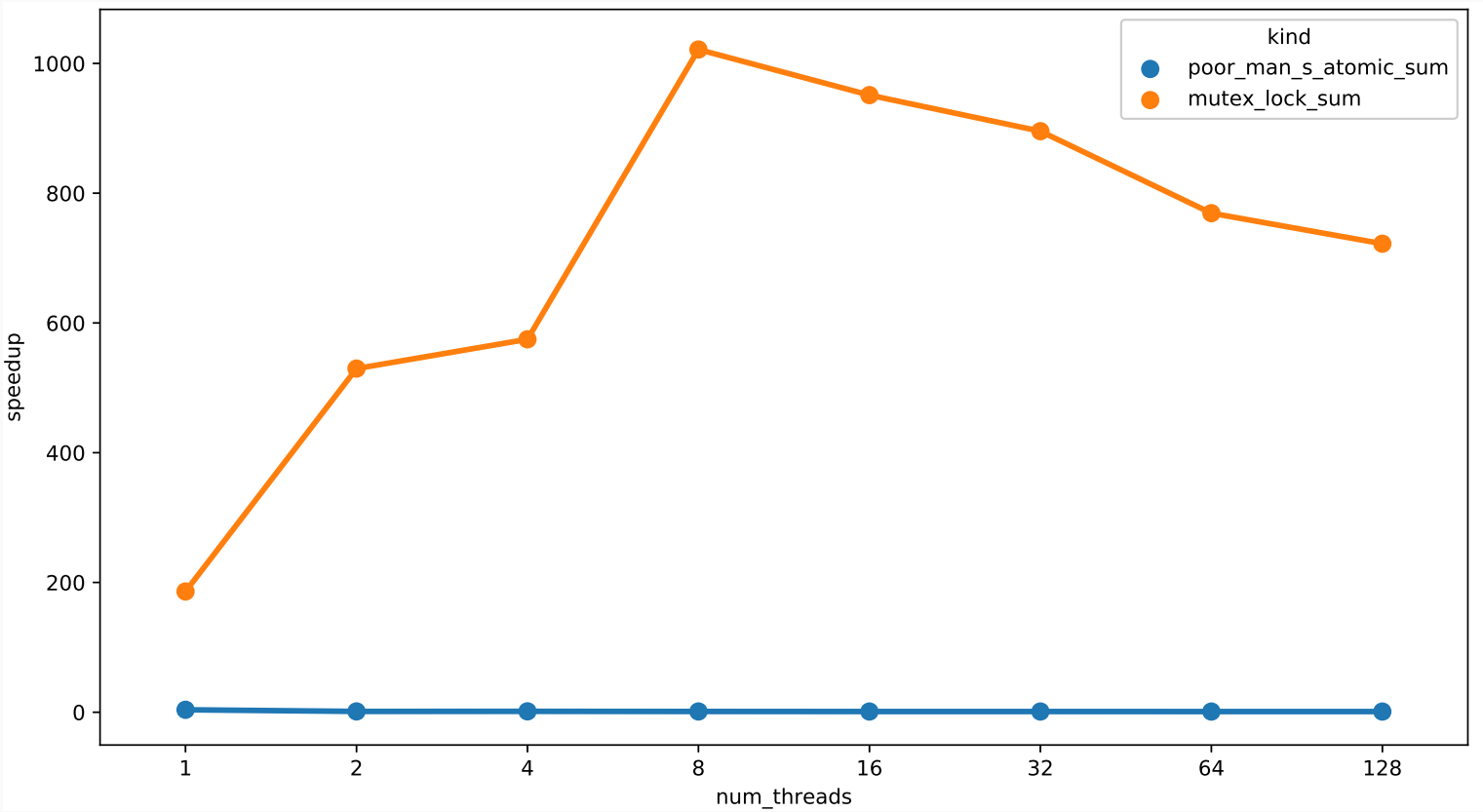



**Figure 1:** 4-core Sum Benchmark

# Atomics: Why Lock-free? Pros vs. Cons

**Pros:**

1. Concurrency
2. Scalability
3. No Deadlock

**Cons:**

1. Hard to be correct
2. Hard to read, maintain
3. Adding new operation to algorithm usually brings data race

# Atomic Operations

1. `load()`, `store()`, `exchange()`, CAS for all built-in types and user-defined trivial types

2. Increment and decrement for raw pointer types
3. Addition, subtraction and bitwise operations for integers (`++`, `--`, `-`, `+=`, `-=`, `&=`, `|=`, `^=`)
4. `std::atomic<bool>`, `std::atomic<double>` are valid, but no special functions

5. Special `std::atomic_flag`, guaranteed to be lock-free

# Atomic Operations, Cont'd

## Compare And Swap Operation (CAS)

Presudo Code for CAS:

```
x.compare_exchange(old_x, new_x);

// is the same as

if (x == old_x) {
  x = new_x;
  return true;
} else {
  old_x = x;
  return false;
}
```

# Atomic Operations, Cont'd

**Compare And Swap Operation (CAS)**

C++ Provides two versions of CAS (weak and strong)

1. compare_exchange_strong works as the presudo code
2. compare_exchange_weak works as strong, but may "spuriously fail", which may return false even if `x == old_x`

# Optimization, Races, Perspectives from Hardware and Software

# What programmers think of his/her machine

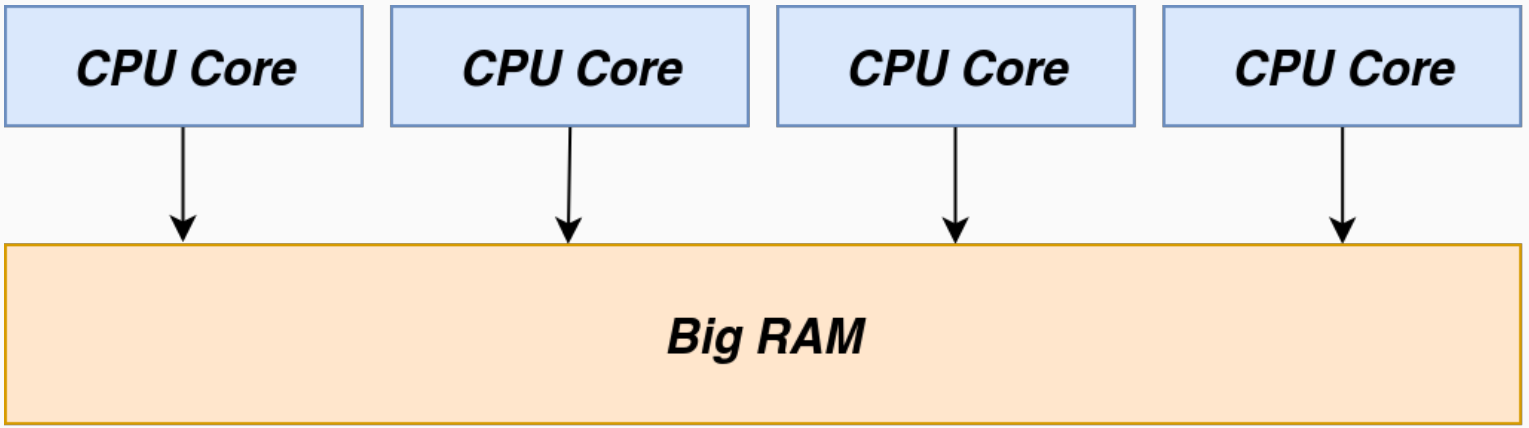



**Figure 2:** Machine Model In Illusion

# What machines are actually alike

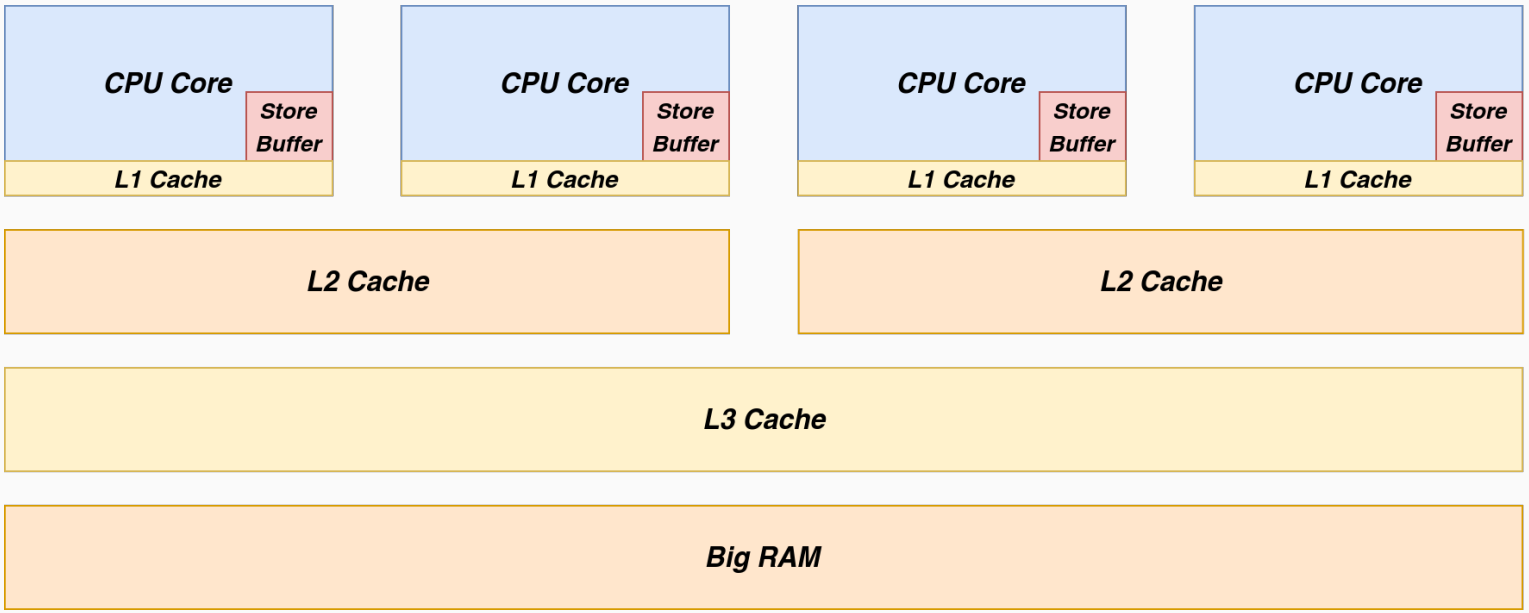



**Figure 3:** Machine Model In Reality

# Optimizations

# Optimizations

## Compiler Optimizations

# Optimizations

**Compiler Optimizations**

**Processor Out-of-order Execution**

# Optimizations

**Compiler Optimizations**

**Processor Out-of-order Execution**

**Cache Coherency**

# Dekker's and Peterson's Algorithm

## Initialize

```
bool flag0 = false;
bool flag1 = false;
```

## Thread 1

```
flag0 = true;
while (flag1 == true) {
  // busy wait
}
// enter critical section

// ... process ...

// exit critical section
flag0 = false;
```

## Thread 2

```
flag1 = true;
while (flag0 == true) {
  // busy wait
}
// enter critical section

// ... process ...

// exit critical section
flag1 = false;
```

# Dekker's and Peterson's Algorithm, Cont'd

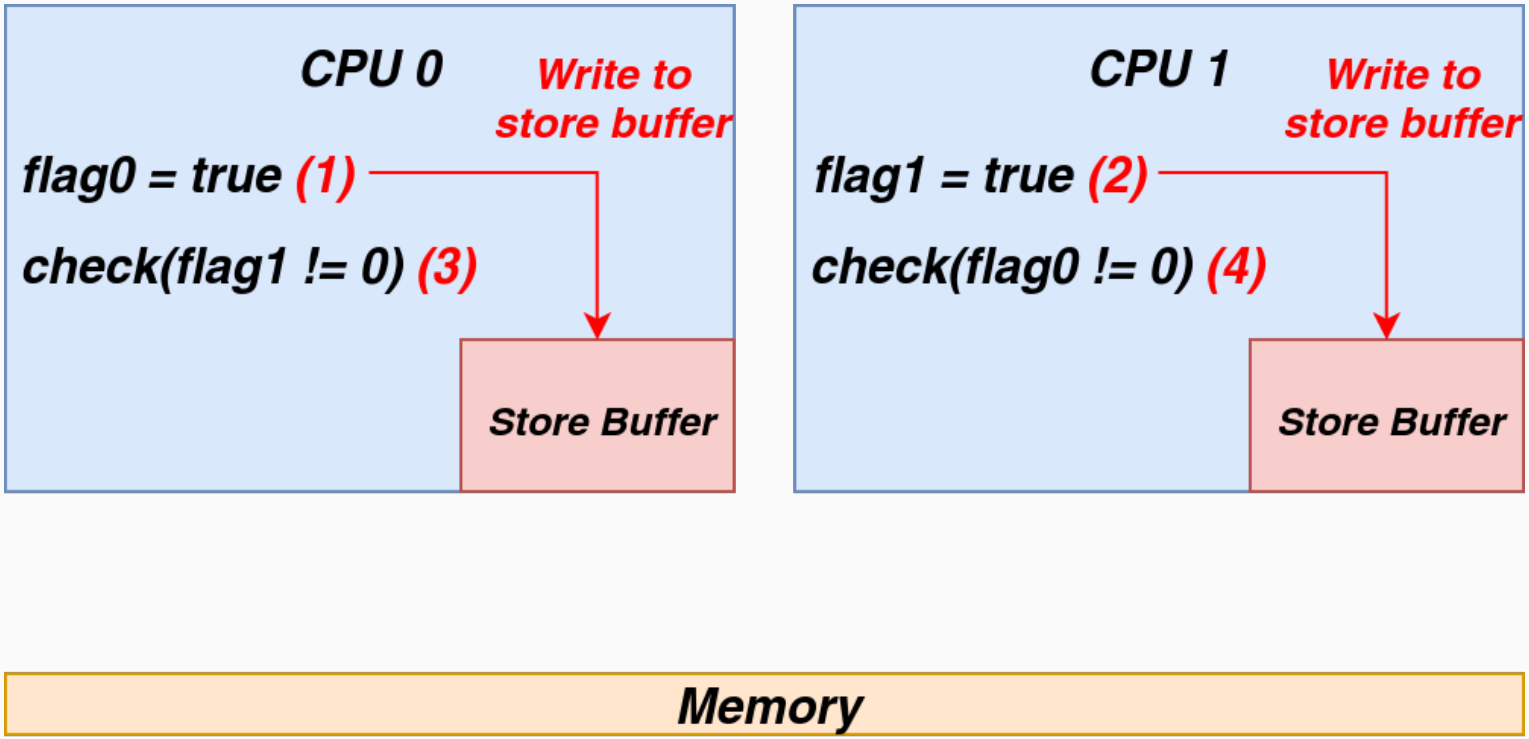
**Q1: Could it be possible that both threads enter critical section?**

# Dekker's and Peterson's Algorithm, Cont'd

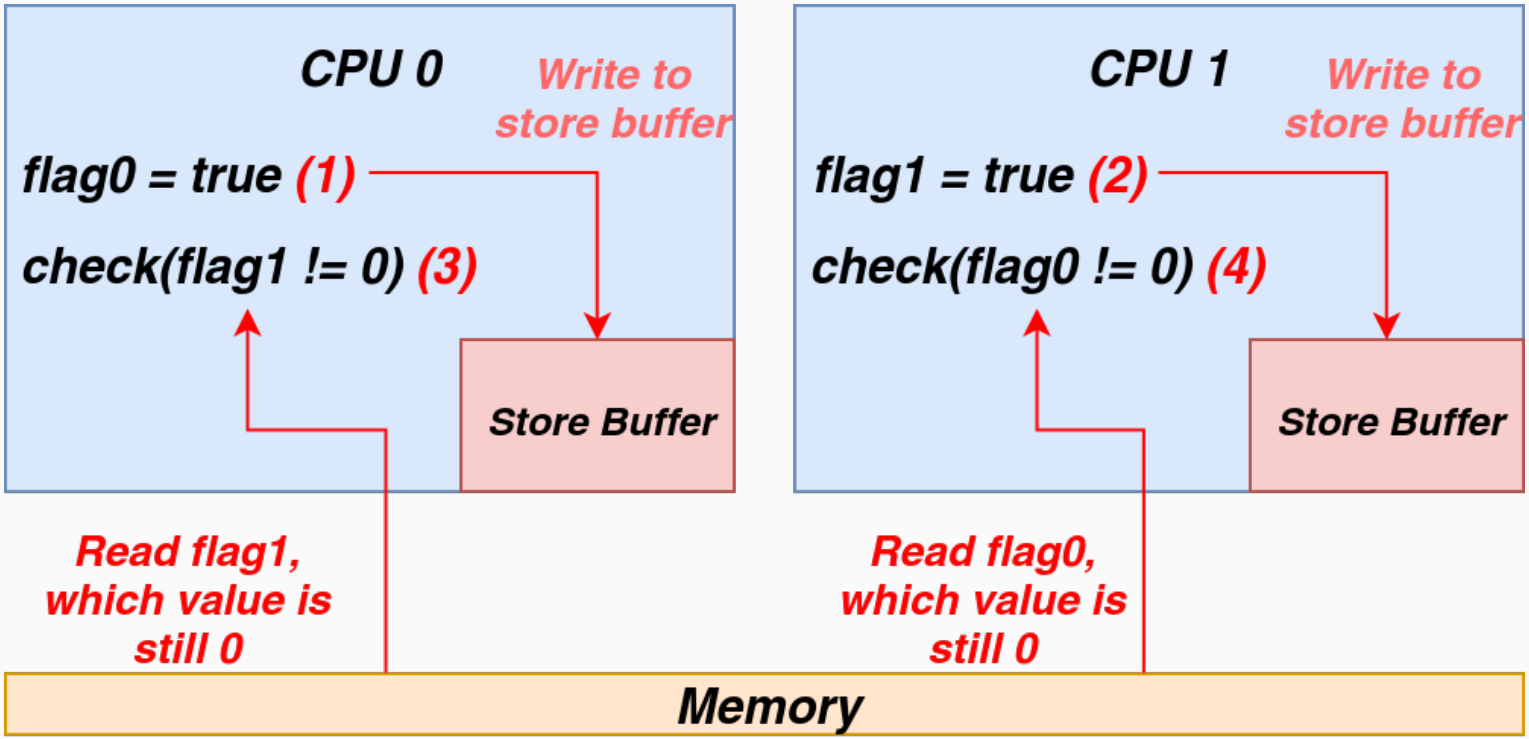
**Q1: Could it be possible that both threads enter critical section?**

**Q2: Assume no compiler reordering or out-of-order execution. Q1?**

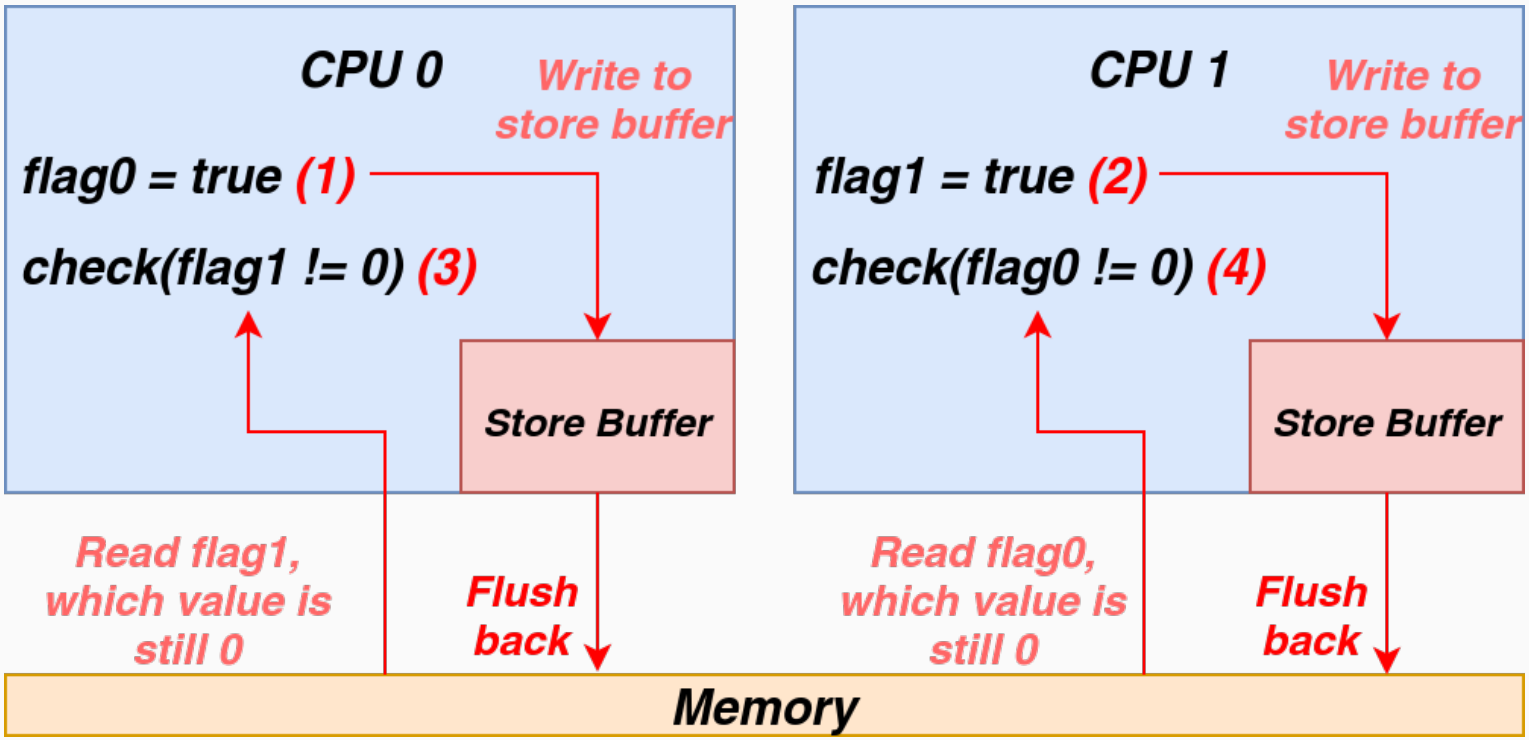
# Dekker's and Peterson's Algorithm, Cont'd



# Dekker's and Peterson's Algorithm, Cont'd



# Dekker's and Peterson's Algorithm, Cont'd



# Make No Assumptions

1. Set flag0

2. Then Check flag1

# Make No Assumptions, Cont'd

**Always measure performance before optimization**

**Always measure again to ensure optimization does give you some gain**

# Single-Thread Optimizations (Compiler's View)

**From**

```
x = 1;
y = "hello world";
x = 2;
```

**To**

```
y = "hello world";
x = 2;
```

# Single-Thread Optimizations (Compiler's View), Cont'd

From

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

To

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

# Single-Thread Optimizations (Compiler's View), Cont'd

**From**

```
x += 1
y -= 2
z += 3
```

**To**

```
z += 3
y -= 2
x += 1
```

# Single-Thread Optimizations (Compiler's View), Cont'd

From

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

To

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

# Single-Thread Optimizations (Compiler's View), Cont'd

**What the compiler knows?**

1. All memory operations in this thread and exactly what they do, including data dependencies
2. How to be conservative enough in the face of possible aliasing

**What the compiler does not know?**

1. Which memory locations are "mutable shared" variables and could change asynchronously due to memory operations in another thread
2. How to be conservative enough in the face of possible sharing

# Single-Thread Optimizations (Compiler's View), Cont'd

**Solution: Tell it**

Identify the operations on "mutable shared" locations, i.e. mutex, atomics, …

# Sequential Consistency (SC)

**Defined in 1979 by Leslie Lamport:**

> *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 order specified by its program*

**Keeping this illusion can be very expensive**

# Sequential Consistency (SC), Cont'd

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

1. Java: SC-DRF required since 2005
2. C11/C++11: SC-DRC by default

# Memory Model == Contract

**Your Promise:**

To correctly synchronize your program (no race conditions)

**”The system” Promise:**

To provide the illusion of executing the program you wrote

# Memory Model == Contract, Cont'd

**Has anyone ever debugged a release build (-O2/-O3) program?**

# Memory Model == Contract, Cont'd

**Has anyone ever debugged a release build (-O2/-O3) program?**

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

# General Acquire and Release Concepts

# Transaction

**Atomic**

Commit or nothing

**Consistent**

Reads a consistent state, or transforms into another consistent state

**Independent**

Correct in the presence of other transactions on same data

# Transaction, Cont'd

## Transaction

```
void transfer(account from, account to, int amount) {
  begin_transaction();  // ACQUIRE exclusively
  from.credit(amount);
  to.debit(amount);
  end_transaction();    // RELEASE exclusively
}
```

# Code Transformation Restrictions (No Moving Out)

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

// Prevented
x = 42;
mutex.lock();
mutex.unlock();

// Also Prevented
mutex.lock();
mutex.unlock();
x = 42;
```

# Code Transformation Restrictions (Ok to Move In)

```
x += 1;
mutex.lock();   // enter critical region (acquire lock)
y = 0;
mutex.unlock(); // exit critical region (release lock)
z += 3;

// OK
mutex.lock();
x += 1;
y = 0;
z += 3;
mutex.unlock();

// But not this:
z += 3;
mutex.lock();
y = 0;
mutex.unlock();
x += 1;
```

# Acquire Release as One-way Barriers

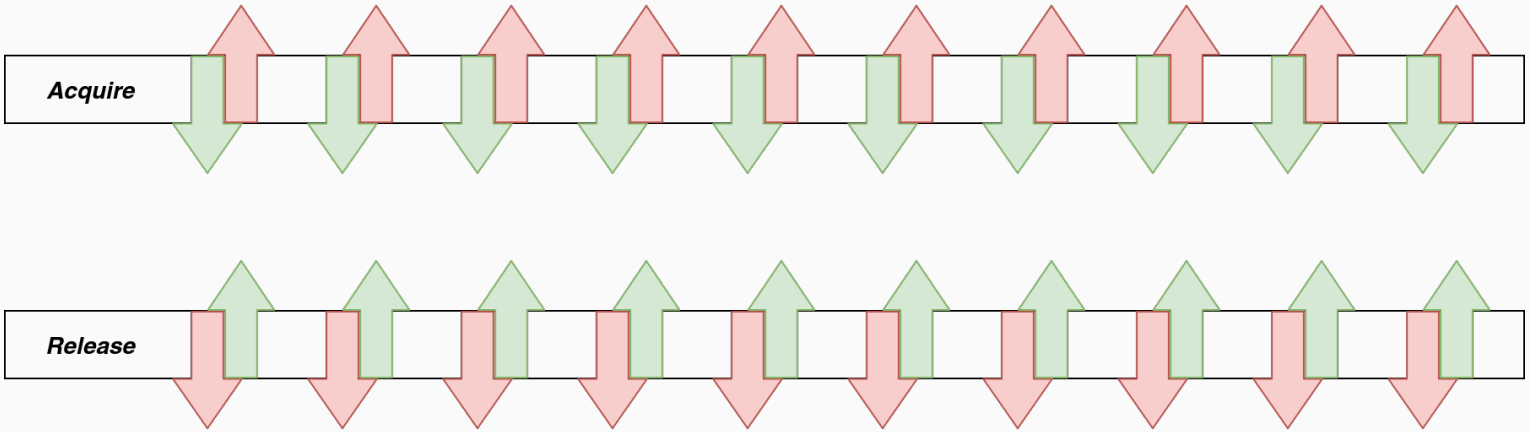



**Figure 4:** Acquire and Release as One-way Barriers

# x86_64 Memory Ordering Model

# x86_64 Memory Ordering Model

1. Reads are not reordered with other reads
2. Writes are not reordered with older reads
3. Writes to memory are not reordered with other writes
4. Two Writes to the same location have a total order (5th rule in manual)
5. Reads, writes are not reordered with locked instructions (6th rule in manual)
6. Locked instructions have a total order

See Intel Architectures Software Developer's Manual, Volume 3 - 8.2. MEMORY ORDERING for details

# x86_64 Memory Ordering Model, Cont'd

**No Guarantee: Loads May Be Reordered with Earlier Stores to Different Locations**

**Want the compiler to generate appropriate memory fences**

# C++ Memory Ordering

# memory_order_relaxed

1. No operation orders memory
2. No reordering for accessing to the same location (free on x86_64)

# memory_order_acquire

1. Reads and writes in current thread are not reordered before this read
2. Previous release writes in other threads to same location (the same atomic variable) is not reordered with this read
3. Basically free on x86_64

# memory_order_release

1. Reads and writes in current thread are not reordered after this write
2. Further reads in other threads may be reordered with this write to different locations (other atomic variables)
3. All writes in the current thread are visible in other threads that acquire read on the same location (the same atomic variable)
4. `atomic::store()` is usually generated with an `xchg` instruction (little cost on x86_64)

# memory_order_acq_rel

1. Only apply on read-modify-write operations (RMW)
2. Reads and writes in current thread are not reordered before or after this RMW
3. All writes in other threads that release the same atomic variable are visible before the modification and the modification is visible in other threads that acquire the same atomic variable

# memory_order_seq_cst

1. As definition
2. Note: As soon as atomic operations on the same location (the same atomic variable) is not synchronized with sequential consistent, the sequential consistency is lost
3. Usually by adding mfences after store operations (very expensive on x86)

# memory_order_consume (Not talked)

# Memory Ordering Performance

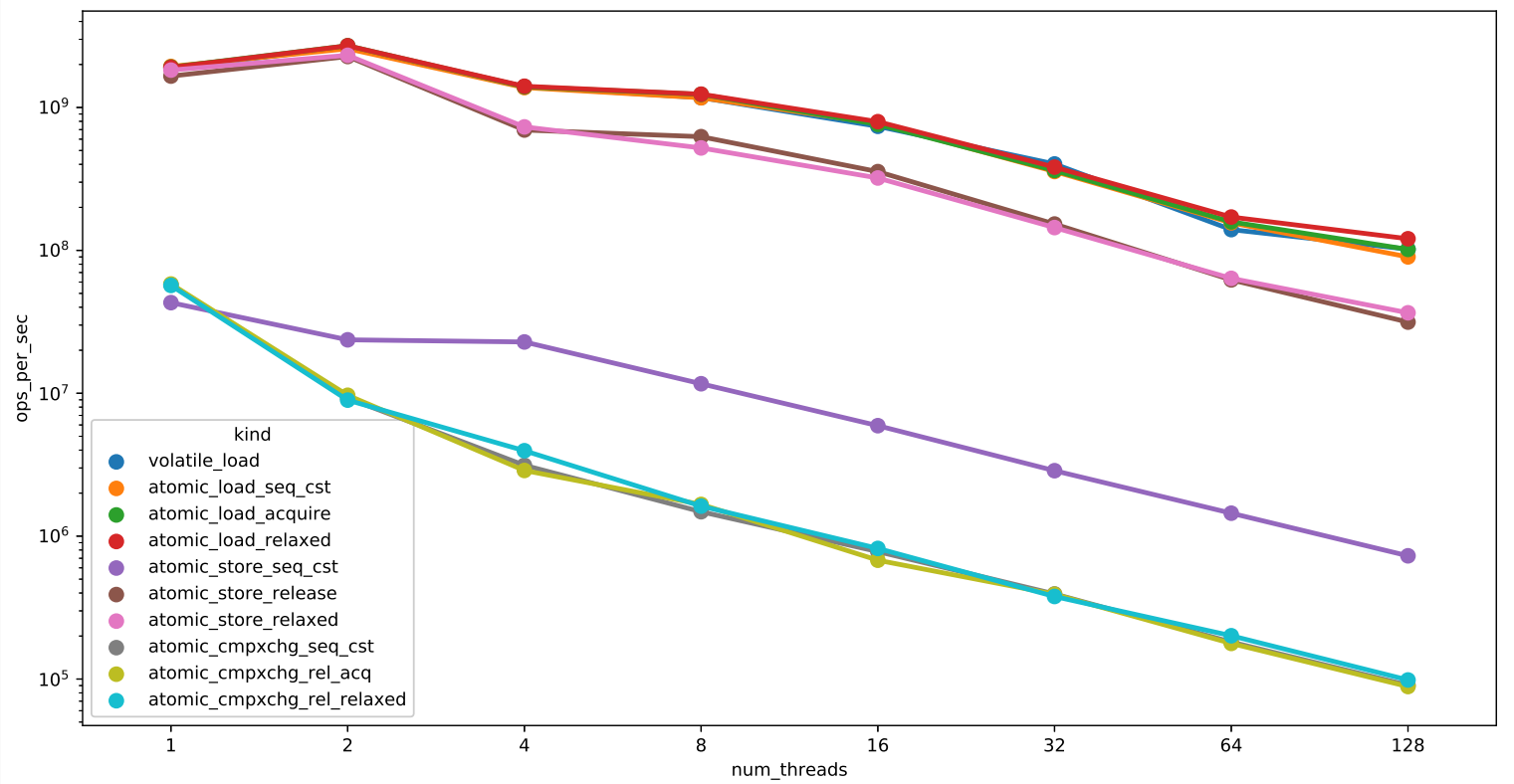



**Figure 5:** 4-core Atomic Operation Performance Graph

# Atomic Thread Fence

# Release Fence - Acquire+ Atomic Synchronization

**Non-formal explain:**

All non-atomic and relaxed atomic stores before the fence will be synchronized with corresponding loads in other threads after the fence is evaluated

# Acquire Fence - Release+ Atomic Synchronization

**Non-formal explain:**

All non-atomic and relaxed atomic stores before the atomic will be synchronized with corresponding loads in other threads after the fence is evaluated

# Release Fence - Acquire Fence Synchronization

**Non-formal explain:**

All non-atomic and relaxed atomic stores before the release fence will be synchronized with corresponding loads in other threads after the acquire fence is evaluated

# C++ Atomics Pitfalls

# Atomic Pitfall No. 1

**`x = x + 1` is not the same as `++x`**

1. Atomic operation is guaranteed individually
2. `x = x + 1` is `x.store(x.load() + 1)`
3. Always avoid AND THEN logics

# Atomic Pitfall No. 2

**Important: Memory order almost always come in pairs**

**Release store() of an atomic can be reordered with SC loads()**

Thread 1

```
x.store(true, std::memory_order_release);
if (!y.load(std::memory_order_seq_cst))
  important_work_1();
```

Thread 2

```
y.store(true, std::memory_order_release);
if (!x.load(std::memory_order_seq_cst))
  important_work_2();
```

**It's a data race!**

# Atomic Pitfall No. 2, Cont'd

Release store() of an atomic can be reordered with SC loads()

By definition, on x86_64 the `mfence` might not be generated after x and y's store operation because it is weakened by the release store

# Atomic Pitfall No. 3

**Mutex-based critical sections does not necessarily guaranteed as a memory fence**

Thread 1

```
x.store(true, std::memory_order_relaxed);
{ std::lock_guard<std::mutex> _(m1); }
if (!y.load(std::memory_order_relaxed))
  important_work_1();
```

Thread 2

```
y.store(true, std::memory_order_relaxed);
{ std::lock_guard<std::mutex> _(m2); }
if (!x.load(std::memory_order_relaxed))
  important_work_2();
```

# Atomic Pitfall No. 3, Cont'd

**Memory Model allows moving into the critical sections**

**Note: Using the same mutex would work**

# Atomic Pitfall No. 4

**”Dependencies” do not enforce ordering w.r.t. threads**

## Pitfall No. 4

```
ptr = x.load(std::memory_order_relaxed);
if (ptr == y) { // hardware predicts the condition true
  // compiler transforms to y->field
  result = ptr->field.load();

  // Both loads issued concurrently
  // Second may complete first
}
```

# Some Correct Use Examples

# Single-word Data Structures

If the contents of a single-word data structure are not relied upon for other computations while it is modified, it's OK to use `memory_order_relaxed`.

**Example:**

1. Counter that's only read after threads.join()

```
counter.fetch_add(1, std::memory_order_relaxed)
```

2. Accumulate information in a short bit vector

```
bit_set.fetch_or(1 << elem, memory_order_relaxed)
```

**Note: Caring only about the result. The process is not synchronized.**

# Computing a guess for `compare_exchange`

Sometimes the value of a load just does't affect correctness

**Example:**

```
old = x.load(std::memory_order_relaxed);
while (x.compare_exchage_weak(old, foo(old));
```

This would remain correct if we replaced the first line by `old = 42`

We're clearly not relying on the load value to tell us anything about the state

# Double-Checked Locking

**DCLP is mostly used in frequent reading, but rarely locked writing scenarios.**

1. Yes, not only singletons
2. Normally giving a atomic cheap and weak hint
3. If the hint shows that costly operations are needed
4. Acquire lock now and double check the hint again

# Double-Checked Locking, Cont'd

## Taking Singleton Initialization Problem as an Example

### Slow always lock version

```
static Singleton *sgt = nullptr;
std::mutex sgt_mutex;

Singleton *instance() {
  std::lock_guard<std::mutex> lock(sgt_mutex);
  if (sgt == nullptr)
    sgt = new Singleton;
  return sgt;
}
```

# Double-Checked Locking, Cont'd

## How to reduce lock contention time

```
static std::atomic<Singleton *>sgt {nullptr};
std::mutex sgt_mutex;

Singleton *instance() {
  Singleton *tmp = sgt.load(std::memory_order_acquire);
  if (tmp == nullptr) {
    std::lock_guard<std::mutex> lock(sgt_mutex);
    tmp = sgt.load(std::memory_order_relaxed);
    if (tmp == nullptr) {
      tmp = new Singleton;
      sgt.store(tmp, std::memory_order_release);
    }
  }
  return sgt;
}
```

# Double-Checked Locking, Cont'd

**But, for singletons in C++11, you should almost always write like the following:**

Singleton

```
Singleton *instance() {
  static Singleton inst;
  return &inst;
}
```

Thank You！