# Foundations of the C++ Concurrency Memory Model

**John Mellor-Crummey**

**Department of Computer Science**
**Rice University**

**johnmc@rice.edu**

# Before C++ Memory Model

- **Prior practice**
  - —threaded programming within a shared address space, e.g. Pthreads
  - —implementations prohibit reordering memory operations with respect to synchronization operations
    - treat synchronization operations as opaque
- **Problem**
  - —C and C++ are single threaded languages using thread libraries
    - language is unaware of threads
  - —compilers also thread unaware
    - optimize programs for a single thread
    - problem: may perform optimizations that violate intended meaning of multithreaded programs

# A Familiar Example

Initially X=Y=0

| T1 | T2 |
|---|---|
| R1 = X | R2 = Y |
| If(R1 ==1) | If(R2 ==1) |
| Y=1 | X=1 |

Is R1=R2=1 allowed?

Without precise memory model definition: yes

How?

T1 T2 speculate values of X and Y respectively as 1 validating other's speculation

# Structure Fields and Races

```
struct s {char a; char b} x;
```

| Thread 1: | Thread 2: |
| --- | --- |
| x.a = 1; | x.b = 1; |

Thread 1 not same as:

```
struct s tmp = x;
tmp.a = 1;
x = tmp;
```

The point: Compilers must be aware of threads

# Register Promotion

## Original Code

```
for (...) {
    ...
    if (mt)
        pthread_mutex_lock(...);
    x = ... x ...
    if (mt)
        pthread_mutex_unlock(...);
}
```

## Optimized Code

```
r = x;
for (...) {
    ...
    if (mt) {
        x = r;
        pthread_mutex_lock(...);
        r = x;
    }
    r = ... r ...
    if (mt) {
        x = r;
        pthread_mutex_unlock(...);
        r = x;
    }
}
x = r;
```

# Why a C++ Memory Model

- **Problem: Hard for programmers to reason about correctness**
- **Without precise semantics, hard to reason if compiler will violate semantics**
- **Compiler transformations could introduce data races without violating language specification**
  - **—resulting executions could yield unexpected behaviors**

# Trylock and Ordering

**Problem: Undesirably strong semantics for programs using non-blocking calls to acquire lock**

- **—e.g., pthread_mutex_trylock()**

```
T1              T2
X=42;           while(trylock(l)==success)
lock(l);          unlock(l);
                assert(x==42);
```

# The Problem with Trylock

```
T1              T2
X=42;           while(trylock(l)==success)
lock(l);          unlock(l);
                assert(x==42);
```

- **Problem: above program is data−race free by current semantics**
- **Assertion can fail if compiler or hardware reorders two statements executed by T1**
- **Prohibiting such reordering requires memory fence before lock**
  - **—fence doubles cost of the lock acquisition**
- **For well−structured uses of locks and unlocks**
  - **—such reordering is safe and no fence is necessary**

# C++ Memory Model Goals

- Sequential consistency for race free programs
- No semantics for programs with data races
  - —different from Java  How?  Why?
- Weakened semantics for trylock

# Why Undefined Data Race Semantics?

- **There are no benign data races**
  - **—effectively the status quo**
  - **—little to gain by allowing races other than allowing code to be obfuscated**
- **Giving Java-like semantics to data races my greatly increase cost of some C++ constructs**
- **Compiler optimizations often assume objects do not change unless there is an intervening assignment through a potential alias**

```
unsigned i = x;
if (i < 2) {
        foo: ...
        switch (i) {
                case 0: ....; break;
                case 1: ....; break;
                default: ....;
        }
}
```

# Possible Memory Models

- **Sequential consistency**
  - —intuitive, but restricts optimizations
- **Relaxed memory models**
  - —allow hardware optimizations
  - —specified at a low level making it hard to reason about correctness for programmers
  - —limits compiler optimizations
- **Data−race free model**
  - —properties
    - guarantee sequential consistency to data−race free programs
    - no guarantee for programs with races
  - —simple model with high performance

# Definitions

- **Memory location**
  - —each scalar value occupies a separate memory location
    - except bitfields inside the same innermost struct or class
- **Memory action consists of**
  - —type of action
    - synchronization operation (for communication)
      lock/unlock, atomic load/store, atomic read-modify-write
    - data operation: load, store
  - —label identifying program point
  - —values to be read and written

# Definitions

- **Thread Execution**
  - —set of memory actions
  - —partial order corresponding to the sequenced before ordering
    - – sequenced before applies to memory operations by same thread
- **Sequentially Consistent Execution**
  - —set of thread executions
  - —total order, <T, on all the memory actions which satisfies the constraints
    - – each thread is internally consistent
    - – T is consistent with sequenced-before orders
    - – each load, lock, read-modify-write operation reads value from last preceding write to same locations according to <T
- **Effectively requires total order that is interleaving of individual thread actions**

# Definitions Continued

- Two memory operations conflict if
  - —they access same memory location
  - —at least one of operation is a
    - store
    - atomic store
    - atomic read-modify-write
- Type 1 data race
  - —in sequentially consistent operation, two memory operations from different threads form a type 1 data race if they conflict
    - at least one is a data operation
    - adjacent in <T

# C++ Memory Model

- **If a program (on a given input) has a sequentially consistent execution with a (type 1) data race, its behavior is undefined**
- **Otherwise, program behaves according to one of its sequentially consistent executions**

# Legal Reorderings

**Hardware and compilers may freely reorder memory operation M1 sequenced before memory operation M2**

**if the reordering is allowed by intra-thread semantics and**

1. **M1 is a data operation and M2 is a read synchronization operation**
2. **M1 is a write synchronization and M2 is a data operation**
3. **M1 and M2 are both data with no synchronization sequence-ordered between them**

# Legal Lock Optimizations

- When locks and unlocks are used in “well-structured” ways, following reorderings between M1 sequenced-before M2 are safe
  - —M1 is data and M2 is the write of a lock operation
  - —M1 is unlock and M2 is either a read or write of a lock
- Note: Data writes and writes from well-structured locks and unlocks can be executed non-atomically

# Trylock Revisited

- **Problem: trylock allows one to infer that a lock has been acquired**
- **Want to prevent this to avoid the cost of and additional fence**
- **Previous work achieved this by distinguishing between different types of synchronization operations and/or redefining data operations**
  - **—data-race-free-1 model distinguishes pairable and unpairable synchronization**
  - **—Java has happens before**

# C++ Memory Model Solution for Trylock

- Modify specification of trylock to not guarantee that it will succeed if the lock is available
- Failed trylock doesn't tell you anything reliable
  - —can't use trylock to infer that another thread holds the lock
- New semantics
  - —successful trylock is treated as lock()
  - —unsuccessful trylock() is treated by memory model as no-op

# Implications for Current Processors

- Atomic writes need to be mapped to xchg on AMD64 and Intel64
- Hardware now only needs to make sure that xchg writes execute atomically
- xchg implicitly ensures semantics of a store|load fence
- Why is this OK?
  - —better to pay a penalty on stores than on loads
  - —store|load fence replaced by read-modify-write is as expensive on many processors today

# Atomic Ambiguity

- **Atomic can either be**
  - **—description of how a write is executed in hardware**
  - **—C++ qualifier to denote special class of memory operations**
- **Default C++ atomic operator needs to be executed atomically by hardware**

# Some Problems

- Significant restrictions on synchronization operations
- Synchronization operations must appear sequentially consistent with respect to each other
- Performance problem in practice
  - —only Itanium distinguishes between data and sync operations
  - —other processors enforce ordering through fence or memory barrier instructions
- Atomic operations must execute in sequenced-before and atomic writes must execute atomically
  - —read can’t return a new value for a location until all older copies of a location are invalid
  - —with caches, atomic writes are easier with invalidation protocols

# Problematic Examples

- Common to use counters that are frequently incremented but only read after all threads complete
  - —problem: requires all memory updates performed prior to counter update become visible after any later counter update in another thread
- Atomic store requires two fences
  - —one before and one after the store
  - —could imagine example where memory updates before or after store could be reordered

# Case for Low-Level Atomics

- Cases exist where current model provides too much safety
- Expert programmers need way to relax model when code does not require as much safety to maximize performance
- Don’t want to make memory model hard to reason about for the rest of us

# Additional Definitions

- **Happens-before**
  - —if a is sequenced before b, then a happens before b
  - —if a synchronizes with b, then a happens before b
  - —if a happens before b and b happens before c, then a happens before c
- **Type-2 data race**
  - —two data conflicting accesses to the same memory location are unordered by happens before

# Modified C++ Memory Model

- If a program(on a given input) has a consistent execution with a (type 2) data race, then its behavior is undefined
- Otherwise, program(on the same input) behaves according to one of its sequentially consistent executions

# Properties

- Type 2 data race
  - —if two data accesses to the same memory location are unordered by happens before
- This model provides sequential consistency to programs whose consistent executions do not contain a type 2 data race
- If a program allows a type 2 data race in a consistent execution, then there exists a sequentially consistent execution, with two conflicting actions, neither of which happens before the other
- A program allows a type 2 data race of a given input iff there exists a sequentially consistent execution in which two unordered conflicting actions are adjacent in the sequential interleaving

# Low-Level Atomics Support

- Memory model provides low-level atomic operations
- Expert programmer can maximize performance
- Atomic variables can be explicitly parameterized with respect to memory ordering constraint x.load(memory_order_relaxed) allows instruction to be reordered with other memory operations.
  - —load is never an acquire operation, hence does not contribute to the synchronizes-with ordering
- For read-modify-write operations, programmer can specify whether the operations acts as an acquire, a release operation, neither (relaxed), or both

# Java/C++ Comparison

- **Primary goal of Java is safety and security**
  - —make large effort to ensure semantics of codes
- **C++ focus is performance**
  - —no such safety concerns
    - ignore semantics of codes with data races
  - —low-level atomics to allow performance fine tuning

# Summary

- Need to write parallel programs to get performance gains on modern architectures
- Previously, C++ memory model was ambiguous
- C++ memory model provides
  - —well defined semantics for data race free programs
  - —high performance by leaving programs with data races undefined
  - —low level atomics for expert programmers to squeeze out maximum performance
  - —compilers may assume that ordinary variables don't change asynchronously
- Remaining problems: adjacent bitfields