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 procedure calls
- **Problem**
  - —C and C++: single threaded languages using thread libraries
    - – unaware of threads
  - —compilers are thread unaware
    - – optimize programs for a single thread
    - – problem: may perform optimizations that are valid for single-thread programs but violate intended meaning of multithreaded programs
  - —prior informal specifications don't precisely indicate
    - – what is a data race
    - – what is the semantics of a program without a data race

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 a precise memory model: yes!

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

Does this program have a data race?

yes, for the aforementioned execution

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 the compiler will violate semantics**
- **Compiler transformations could introduce data races without violating language specification**
  - **—e.g., previous register promotion and field update examples**

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);
```

What's odd about with this code?

T2 waits for T1 to acquire lock instead of waiting for lock to be released

Why would we want the assert to succeed?

The Problem with Trylock

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

- **Problem: in a sequentially consistent execution, example is data-race free by current semantics and assertion can't fail**
- **Assertion can fail if compiler or hardware reorders the 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, reordering T1's statements is safe and no fence is necessary**

C++ Memory Model Goals

- Sequential consistency for race free programs
- No semantics for programs with data races
- 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 may greatly increase cost of some C++ constructs**
- **Compilers often assume that 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: makes it hard for programmers to reason about correctness
  - —can limit compiler optimizations
    - e.g., at least one relaxed model disallows global analysis or RRE
- **Data-race free model**
  - —properties
    - guarantee sequential consistency for data-race free programs
    - no guarantee for programs with races
  - —simple model with high performance

Why Data Race Free Models?

- **Simple programmability of sequential consistency**
  - **—any program without data races is guaranteed to execute with sequential consistency**
- **Implementation flexibility of relaxed models**
- **Different data race free models define notion of a race to provide increasing flexibility**
  - **—data-race-free-0: no concurrent conflicting accesses (as for Java)**

Definitions - I

- **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
    - data operation: load, store
    - synchronization operation (for communication)

      lock/unlock/trylock, atomic load/store, atomic read-modify-write
  - —label identifying program point
  - —values to be read and written

Definitions - II

- **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 - III

- Two memory operations conflict if
  - —they access same memory location, and
  - —at least one 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, the 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 lock & unlock 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

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 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
- Why is this useful?
  - —promise sequential consistency with an intuitive race definition, even for programs with trylock

Sequential Consistency vs. Write Atomicity

- **Independent read, independent write doesn't guarantee sequential consistency if writes don't execute atomically**

```
                Initially X=Y=0
T1          T2          T3          T4
X=1         r1=X        Y=1         r3=X
            fence       fence       fence
            r2=Y        X=2         r4=X

r1=1, r2=0, r3=2, r4=1 violates write atomicity
```

- **T1 and T2 are on same node with shared write-through cache**
- **T3 and T4 are on separate nodes**
- **Execution**
  - **—T2 reads T1's value x=1 early, executes a fence and reads old Y**
  - **—T3 writes y=1, executes fence, writes x=2; all writes commit**
  - **—T4 reads x=2, executes fence, now T1's write x=1 commits**
- **Violation of sequential consistency: non-atomic write x=1**

Making C++ Memory Model Usable

- **Problem: a departure from sequential consistency for sync operations can lead to non-intuitive behavior**
- **Approach: retain sequential consistency for default atomics and sync operations**

C++ Atomics

- C++ qualifier denote variables with atomic operations
  - —operations needs to be executed atomically by HW
- Sequentially-consistent atomics: data-race-free models require that all operations on C++ atomics must appear sequentially consistent
- Initially opposed by many HW and SW developers
  - —sufficiently expensive on some processors that an “experts only” alternative was deemed necessary
  - —existing code (e.g. Linux kernel) assumes weak ordering. easier to migrate such code with primitives closer to assumed semantics
- Why was it resolved this way?
  - —models were too hard to formalize and use otherwise

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

Implications for Current Processors

Guaranteeing sequentially consistent atomics

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

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

- Current model provides too much safety for some use cases
- 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

Low-Level Atomics

- Why?
  - —enable expert programmers to maximize performance
- What?
  - —can explicitly parameterize an operation on an atomic variable with its memory ordering constraints
    - e.g., 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 an operation acts as an acquire, release, neither, or both

Additional Definitions

- **Happens-before (HB)**
  - —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

C++ Memory Model for Low Level Atomics

- 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

A theorem shows that this is equivalent to the previous model phrased in terms of type 1 data races

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 enable performance fine tuning

Summary

- Need multithreaded programs to harness the power of modern multicore architectures
- A semantics for multithreaded execution is essential
- 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