Foundations of the C++ Concurrency Memory Model

ACM SIGPLAN Conference on
Programming Language Design and Implementation, 2008

Hans-J. Boehm
(HP Laboratories)

Sarita V. Adve
(University of Illinois)

presented by Nicola Vermes

Introduction

(1) C++: a single-threaded programming language

(2) Pthreads: separate threads library

(3) Parallel applications: combining (1) and (2)

(4) Since (1), compilers are thread-unaware:

- generate code as for single-threaded application
- transformations/optimizations (i.e. reordering)
- do not preserve the meaning of the multithreaded program

(5) Hardware: further transformations

This obviously makes difficult to reason about parallel programs

Currently solutions

- Threads libraries provide synchronization primitives (mutex, semaphores, fences, …)
- To prohibit concurrent accesses to shared memory
- But also to apply restrictions for compilers:
  it is prohibited to reordering synchronization operations with respect to ordinary memory operations (in a given thread)

Unfortunately this is insufficient
We need a more formal model for multithread

Why insufficient?

```
   Initially X=Y=0
T1                  T2
r1=X                r2=Y
if (r1==1)          if (r2==1)
    Y=1                 X=1
```

- Is outcome r1=r2=1 allowed? Yes!
- Is this program data-race-free? Unclear
- Possible if compiler reorders the last 2 lines of T1, or if hardware speculates on values of X/Y
- This example shows that is unclear and difficult to reason about parallel programs and data-races

… at least without a formal memory model

Memory model

- specifies what values a read can return
- affects programmability
- affects performance and portability
- must be defined also for any part that transforms the program (compilers, hardware)
- The HW memory model must be consistent with the memory model of the software
- The most intuitive: Sequential Consistency Model
  - Very simple for programmers but very restrictive for optimizations (the main trade-off of memory models)

Sequential consistency

A program is Sequential Consistent if:

- *the result of any execution is the same as if the operations of all threads were executed in some sequence*
- *and the operations of single threads are in the program order*

- Efforts to determine transformations SC-safe, but anyway most of compilers/HW don't preserve SC
- Relaxed model: specified at low level (difficult to reason for programmers) and not so efficient

Data-Race-Free Models

- Class of models with a different approach:
  correct programs are only those w/o data-races
- It guarantees SC to programs w/o data-races
- It does not define anything about the behavior of programs with data-races
- It formalizes the notion of data-race
  - the notion changes depending on the specific model
  - Data-Race-Free-0 model uses the simplest notion
- It combines the simple programming model of SC and good performances

The proposed C++ Model

- A memory action can be of a synchronization type (un/lock, atomic load/store/read-modify-write) or of a data operation type (load, store)
- Thread execution: set of memory actions with a sequenced-before order
- SC execution: set of thread executions with a total order $<_T$ on memory actions
- $<_T$ respects the SB order (thread exec. internally consistent)
- Each load/lock/RMW reads the value from the last preceding (according to $<_T$) write on the same location
- Last operation on a lock preceding an unlock must be a lock operation by the same thread
- Effectively $<_T$ is just an interleaving of thread actions

Data-Race

- Two memory operations conflict if
  1. access the same memory location
  2. and at least one is a store or atomic store/RMW
- In a SC execution two memory operations from different threads form a data-race if:
  1. they conflict
  2. and at least one is a data operation
  3. and they are adjacent in some interleaving with respect to $<_T$ (i.e., they may be executed in parallel)
- If a program has a data race, the behavior is undefined, otherwise behaves according SC

Optimizations allowed

- Compilers can reorder memory operation A sequenced before B if:
  - A: data operation; B: read synchronization operation
  - A: write synchronization op.; B: data op.
  - A and B both data with no synchronization sequence-ordered between them
  - A: data op.; B: write of a lock op.
  - A: unlock; B: is either a read or write of a lock
- At the hardware level, data writes and writes from (well-structured) un/locks can be executed non-atomically

The trylock() issue

```
T1                          T2

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

- It inverts the sense of a lock, but it is sometimes used
- It is DRF, apparently the assertion cannot fail, but it can if the two T1's statements are reordered (optimization allowed)
- Inefficient solutions: avoid reordering with fences; distinguish sync./data op. to detect these situations, consider them races
- A simple solution is new definition of trylock():
  it is not guaranteed that it will succeed if the lock is available (it can “spuriously fail”)
- With this new definition is clear that the assertion may fail: the execution in which it fails is now SC: it simply involved a spurious trylock() failure

Sequentially consistent atomics

- The model requires that synchronization operations appear sequentially consistent, i.e. must be executed in a sequenced-before order
- New values of synchronization variables must be propagated to all threads in the same order
- For single-core/thread processor: directory-based cache coherence protocol
- For multicore: one needs to specific atomic instructions (like CAS or xchg)
- Thus Intel/AMD influenced by this fact and they are now implementing HW with a clear and efficient way to guarantee SC

Low-Level atomics

- Some processors have a mechanism to weaken memory ordering
- Low-Level atomics: instructions that allow to specify (relaxed) memory ordering constraints
- This leads to a more complicated model
- But has been proved that is exactly the same (equivalent) as that presented

Semantics of DR: undefined

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

- x is a shared global variable
- in foo i is spilled (i.e., not keep in register)
- switch uses a branch table
- Compiler reloads value of i from x
- 0<=i<=1: compiler do not check bounds and eliminate the default branch

- if during foo there is a race on x, and its new value is >1, the branch table is accessed out of bounds
- With these compiler's optimizations we can fall on a behavior very hard to define
- The result is a wild branch, i.e. arbitrary code

Conclusions

- Users: simple programming model (don't care about complexity/intricacies of HW memory model)
- Compiler implementors: doesn't change anything relevant
- HW implementors: provide sequential consistency for synchronization operations
- Increasing consensus for “sequential consistency for data-race-free” as the fundamental model

Questions?