Memory Consistency

Don Porter
CSE 506

Logical Diagram

Difficult topic

+ Memory consistency models are difficult to understand
+ Knowing when and how to use memory barriers in your programs takes a long time to master
+ I read the long version of this paper about once a year
+ Started in graduate architecture, still mastering this
+ Even if you can’t master this material, it is worth conveying some intuitions and getting you started on the path
+ Multi-core programming is increasingly common

Background

+ In the 90s, people were figuring out how to build and program shared memory multi-processors
+ Several hardware and compiler optimizations that worked well on single-CPU systems were causing “heisen-bugs” in correct parallel code
+ Disabling all optimizations made this code correct, but slow
+ Various consistency models strike different balances between optimization and programmability

Simple example

/* Pre condition: flag = 0 */

x = a + b
flag = 1

This line is independent of the one above. Execute first, since result is identical

a is in the cache yet
(or ALU is busy, etc)

Extended to multi-processors

/* Pre condition: flag = 0 */

Thread 1

x = a + b
flag = 1

val = x

flag is acting as a barrier to synchronize read of x after x was written

Thread 2

while ( ! flag ) { 1; }
Distinction
+ Compiler/CPU can figure out when instructions can be safely reordered within a given thread
+ Hard to figure out when the order is meaningful to coordinate with other threads
+ If you want optimizations (and you do), programmer MUST give hardware and compiler some hints
  + Hard to design hints that average programmer can successfully give the hardware

Definitions
+ Cache coherence: The protocol by which writes to one cache invalidate or update other caches
+ Memory consistency model: How are updates to memory published from one CPU to another
  + Reordering between CPU and cache/memory?
  + Are cache updates/invalidations delivered atomically?
  + Coherence protocol detail that impacts consistency
  + Distinction between coherence and consistency muddled

Intuition
+ On a bus-based multi-processor system (nearly all current x86 CPUs), a write to the cache immediately invalidates other caches
  + Making the write visible to other CPUs
  + But, the update could spend some time in a write buffer or register on the CPU
  + If a later write goes to the cache first, these will become visible to another CPU out of program order

Sequential Consistency
+ Simplest possible model
  + Every program instruction is executed in order
  + No buffered memory writes
  + Only one CPU writes to memory at a time
    + Given a write to address x, all cached values of x are invalidated before any CPU can write anything else
    + Simple to reason about

Sequential is too slow
+ CPUs want to pipeline instructions
  + Hide high latency instructions
  + Sequential consistency prevents these optimizations
  + And these optimizations are harmless in the common case

Relaxed consistency
+ If the common case is that reordering is safe, make the programmer tell the CPU when reordering is unsafe
  + Details of the model specify what can be reordered
  + Many different proposed models
  + **Barrier (or fence)**: common consistency abstraction
    + Every memory access before this barrier must be visible to other CPUs before any memory access after the barrier
    + Confusing to use in practice
Total Store Order (TSO)

+ Model adopted in nearly all x86 CPUs
+ All stores leave the CPU in program order
+ CPU may load "ahead" of an unrelated store
  + Ex: \( x = 1; y = z \)
+ CPU may load \( z \) from memory before \( x \) is stored
+ CPU may not reorder load and store of same variable
+ Atomic instructions are treated like a barrier

TSO benefits

+ Since nearly all locks involve an atomic write, the CPU will never reorder a critical region with a lock
+ If you use locks, you rarely need to worry about consistency issues
+ When do you worry about memory consistency?
  + Custom synchronization / lock-free data structures
  + Device drivers

5a Example

/* Pre condition: \( A = \) flag1 = flag2 = 0 */

Thread 1
flag1 = 1
A = 1
Register1 = A
Register2 = flag2
Register 1 = 1, R2 = 0, R3 = 2, R4 = 0

Thread 2
flag2 = 1
A = 2
Register3 = A
Register4 = flag1

Both CPUs forward write of \( A \) internally before globally visible

Reorder
Load of R2, R4 ahead of stores

5a Example + barriers

/* Pre condition: \( A = \) flag1 = flag2 = 0 */

Thread 1
flag1 = 1
A = 1
barrier
Register1 = A
Register2 = flag2

Thread 2
flag2 = 1
A = 2
barrier
Register3 = A
Register4 = flag1

Must be a sequential ordering of store \( A \)’s

A = 2 and R2 = 0 or A = 1 and R4 = 0; R2 & R4 != 0

5a Example: order 1

/* Pre condition: \( A = \) flag1 = flag2 = 0 */

Thread 1
flag1 = 1
A = 1 (1)
barrier
Register1 = A
Register2 = flag2 (2)
A = 2 and R2 = 0 or A = 1 and R4 = 0; R2 & R4 != 0

Thread 2
flag2 = 1
A = 2 (3)
barrier
Register3 = A
Register4 = flag1

5a Example: order 2

/* Pre condition: \( A = \) flag1 = flag2 = 0 */

Thread 1
flag1 = 1
A = 1 (1)
barrier
Register1 = A
Register2 = flag2

Thread 2
flag2 = 1
A = 2 (2)
barrier
Register3 = A
Register4 = flag1 (2)

A = 2 and R2 = 0 or A = 1 and R4 = 0; R2 & R4 != 0
Summary

- Identifying where to put memory barriers is hard
- Takes a lot of practice and careful thought
- Looks easy until you try it alone
- But, CPUs would be super-slow on sequential consistency
- Understand: Why relaxed consistency? What is TSO?
  Roughly when do developers need barriers?
- Advice: Take grad architecture; read this paper yearly