6.824 2006 Lecture 11: Memory Consistency (2)

Review from previous lecture:
  We want to make it possible to write correct parallel/distributed programs.
  We assume different CPUs interact only through a storage system.
    Memory, distributed shared memory, or a file system.
  So we need a "memory consistency model"
    That tells us what to expect when we read/write memory.
  We want a model that:
    Is easy to understand, so programmers can easily write correct programs.
    Is possible to implement efficienctly.

One reasonable model: sequential consistency
  Is an execution (a set of operations) correct?
  There must be some total order of operations such that
  1. all CPUs see results consistent with that total order
     i.e. reads see most recent write in the total order
  2. each CPU's instructions appear in-order in the total order

Intuitive justification:
  The single total order means it's easy for one CPU to predict what other CPUs will see
  The "consistent with" and lack of real time may make it easy to implement
  The system appears free to interleave instruction streams however it likes
    to form the total order
  However! When executing in real time, once the system reveals
    a written value to a read operation, the system has committed to a
    little bit of partial order. this may have transitive effects.
  So in real life the system only has freedom in ordering more or
    less concurrent operations -- ones that haven't been observed yet

Remember our mutual exclusion example:
  CPU0: x = 1; if(y == 0) { critical section; }
  CPU1: y = 1; if(x == 0) { critical section; }
  We want this to work.

Lay out style of argument
  there is more than one legal result, depending on interleaving! (not like uniprocessor)
  typical question: is xxx a correct result under sequential consistency?
    "yes" if you can demonstrate an interleaving that gets that result
    "no" if you can show no interleaving could get that result
  main example:
  CPU0: w(x)0   w(x)1 r(y)?
  CPU1: w(y)0   w(y)1 r(x)?
  we can evaluate all legal seq consistency interleavings manually:
    1/1? 1/0? 0/1? 0/0? [only 0/0 is illegal]

Good: sequential consistency causes our example to have intuitive results

How can we implement sequential consistency?

```
straw man 1:
  internet cloud, hosts
  assume each host has a big cache
    and that all data is cached on every host
  reads are local, so they are very fast
  send write msg to each other host
    (but don't wait)
  what goes wrong?
    CPU0 starts the x=1 write, and CPU1 starts y=1 write.
      I.e. they send a packet on the network.
    Both read before their write is visible
    So they both read 0 and enter the critical section.
    i.e. read is before write in total order
    this violates Rule 2
  Lesson: each CPU must wait for each operation to complete.

straw man 2:
  we can achieve per-CPU order by changing write:
    write local cache
    send write msgs to other CPUs
    wait for ACKs from all other CPUs
    only then proceed to instruction after the write
  this fixes our mutex example
    if CPU0's r(y) = 0, then CPU0 has not sent write ACK for
      CPU1's w(y)1, so CPU1 has not executed r(x).
  what goes wrong?
    turns out we need a new example
    CPU0: w(x)1  r(x)?
    CPU1: w(x)2  r(x)?
    legal seq consistency results? 1/1 2/2 1/2 BUT NOT 2/1
    2/1 would violate rule 1
  can we get 2/1 w/ straw man 2 implementation?
    yes: if both write local cache, then wait for remote write ACK.
    more generally, if writes arrive in different orders on different
CPUs
  Lesson: for each memory location, execute operations one at a time

These two rules are sufficient to implement sequential consistency:
  1. Each CPU to execute reads/writes in program order, one at a time
  2. Each memory location to execute reads/writes in arrival order, one
at a time
  proof in Lamport 1979

What kind of implementation would fit well with these rules?
  Single entity in charge of ordering each CPU's operations (i.e. the
CPU).
  Single entity in charge of ordering each location's operations.
  You don't need a central entity to choose the single total order!
  Example: partition memory over multiple modules on a network.
    Send all memory ops to relevant module.
    Divides up memory load nicely for good parallelism.

Does your lab 5 enforce sequential consistency for i-node blocks?
  Each machine's operations on an i-node are ordered (due to lock
client code...)
```

```
  Lock server serializes read and write ops on an i-node by different
machines.
  Lots of details: e.g. wait for block server reply before releasing
lock.
  But across i-nodes: no! ccfs issues concurrent operations.

So you don't *need* to send all memory operations to home module.
  Home module can grant ownership using tokens or locks.

To what extent can you optimize sequential consistency?
  Delegate ownership via tokens
    So home module is serializing token grants, not memory operations
    Memory operations execute (mostly) in local caches
    This makes single-writer workloads fast
  Shared read caching also works
    CPUs cannot tell there was no global total order for reads
    Still need to serialize writes through home module
  But you can't make both reads and writes fast, in general
    Because memory system has to serialize operations for each location
    Which requires communication

In what sense is sequential looser than strict?
  I.e. what are we giving up?
  I.e. what programs will break?
  Answer: seq const doesn't let you reason about timing.

In general seqential consistency doesn't let you reason based on real
time
  CPU0: w(x)0  w(x)1
  CPU1: w(y)0       w(y)2
  CPU2:                  r(y)? r(x)?
  Suppose observer knows operations occured in this temporal order
  Strict consistency requires r(y)1 r(x)2
  But sequential consistency allows either or both to read as zero
  You *can* reason based on per-CPU instruction order and observed
values:
    e.g. CPU1: if(x==1)y=2
    then r(y)2 => r(x)1
    because w(x)1 must have finished before r(x) starts

Example of a faster consistency model?
  We're willing to accept more work for the programmer.
    Though we still want a well-defined model.
  And in return we expect faster execution.

Release Consistency
  You rarely see programs like the a=1; if(b==0) example.
    Because it's so hard to reason about them.
  Instead, parallel programs typically lock data that is shared and
mutable.
    To create atomic multi-step sequences.
    (Not the same as cache ownership tokens...)
  Example: bank account transfer:
    acquire(l);
    b1 = b1 + x;
    b2 = b2 - x;
    release(l);
```

```
  Other CPUs aren't allowed to look at b1 or b2 while l is locked.
  So CPU could do the operations in any order within the critical
section.
    I.e. load b2 before storing b1.
  Rules:
    1. CPU can't re-order any LD/ST before the acquire().
       (otherwise you might read b1 while someone else has the lock)
    2. Writes must finish before release() completes.
       (otherwise other CPUs might not see the writes)
  Can re-order, cache, &c within release/acquire, so fast.
  But: memory system must understand locks, acquire(), and release().

The Treadmarks paper is all about implementing release consistency.
```