6.824 2006 Lecture 9: Memory Consistency (1)

Replicated data a huge theme in distributed systems
  For performance and fault tolerance
  Often easier to replicate data than computation (Hypervisor...)
  Examples:
    Replicated mailboxes and user info in Porcupine.
    Caches in NFS and Echo.
    File server in labs 4 and 5.
    shared-memory multiprocessors
  All these examples involve sophisticated optimizations for performance
  How do we know if an optimization is correct?
  We need to know how to think about correct execution of distributed programs.
  Most of these ideas from multiprocessors 20/30 years ago.
    So I'll talk about memory, loads, stores.
    But ideas are similar for e.g. lab 5
  For now, just correctness and efficiency, not fault-tolerance.

What is "correct" for uniprocessor programs?
  First, define "correct" for each instruction independently
    takes the machine from one state to another
    e.g. ADD R1, R2, R3
    or LD gets value of last ST to same address
  Rule: a result is "correct" if it's the same as a result obtainable by:
    Executing the instructions one at a time, waiting for each to complete

Uniprocessor correctness is a useful definition because programmer can use it
  to predict what program will do, and thus write correct programs
  you can tell this is a *definition* because modern
    CPUs don't work like this; they break this rule;
    lots of logic to ensure they *look* like they are enforcing it

Example of why CPU doesn't want to *implement* the uniprocessor rule:
  MUL R1, R2, R3
  ST x, R1
  LD y, R4
  MUL is pretty slow, ST has to wait for it.
    Dependency via R1.
  But LD does not need to wait: will get same result if we execute it early.
  So generally the LD executes before the ST, for speed.
  But CPU h/w checks if &x == &y, stalls LD if same memory address.
  Point: this optimization only possible w/ a definition!

What about correctness for distributed computations?
  multiple hosts, shared memory
  memory could be files, DSM (next paper), or DHT

Naive distributed memory:
  internet cloud, hosts CPU0, CPU1, CPU2
  assume each host has a local copy of all of memory
  reads are local, so they are very fast

```
  send write msg to each other host
    (but don't wait)

Example 1:
  Simple mutual exclusion algorithm, for locking.
  x and y start as zero on both CPUs
  CPU0:
    x = 1;
    if(y == 0)
      critical section;
  CPU1:
    y = 1;
    if(x == 0)
      critical section;
  Intuitive explanation for why this should "work":
    If CPU0 sees y == 0, CPU1 can't have reached "y = 1",
    so CPU1 must see x == 1, so it won't execute critical section.
  perhaps neither will enter, but never both

Example 1 fails w/ naive distributed memory (and on most multi-
processors).
  Problem A:
    CPU0 sends write x=1 msg, reads local y=0
    CPU1 reads local x=0 before write msg arrives
    local memory and slow writes cause disagreement about r/w order
      CPU0 thinks its x=1 was before CPU1's read of x
      CPU1 thinks its read of x was before arrival of x=1
    so both can enter the critical section!

Example 2:
  CPU0:
    v0 = f0();
    done0 = true;
  CPU1:
    while(done0 == false)
      ;
    v1 = f1(v0);
    done1 = true;
  CPU2:
    while(done1 == false)
      ;
    v2 = f2(v0, v1);
  Intuitive intent:
    CPU2 should execute f2() with results from CPU0 and CPU1
    waiting for CPU1 implies waiting for CPU0

Example 2 won't work with naive distributed memory:
  Problem B:
    CPU0's writes of v0 and done0 may be interchanged by network
    leaving v0 unset but done0=true
  But assume each CPU sees each other's writes in issue order
  Problem C:
    CPU2 sees CPU1's writes before CPU0's writes
    i.e. CPU2 and CPU1 disagree on order of CPU0 and CPU1 writes

Lesson:
  either naive distributed memory isn't "correct"
```

```
  or we should not have expected Examples to work

How can we write correct distributed programs w/ shared storage?
  Memory system promises to behave according to certain rules.
  We write programs assuming those rules.
  Rules are a "consistency model"
  Contract between memory system and programmer

What makes a good consistency model?
  There are no "right" or "wrong" models
  A model may make it harder or easier to program
    i.e. lead to more or less intuitive results
  A model may be harder or easier to implement efficiently

How about "strict consistency":
  each instruction is stamped with the wall-clock time at which it
started
    across all CPUs
  Rule 1: LD gets value of most recent previous ST to same address
  Rule 2: each CPU's instructions have time-stamps in execution order
  Essentially the same as on uniprocessor

Would strict consistency execute Example 1 intuitively?
  Could both CPUs be in the critical section?
    i.e. could both CPUs read 0?
  I.e. can we show a time-stamp ordering of operations, consistent with
rules,
    that leads to both CPUs in critical section?
  Rule 2 says each CPU's operations occur in time order in execution
order, so
    CPU0: w(x)1 r(y)0
    CPU1: w(y)1 r(x)0
    but we're not sure of interleave
  CPU0's r(y)0 means w(y)1 hadn't executed by Rule 1, so
    CPU0: w(x)1 r(y)0
    CPU1:             w(y)1 r(x)0
  But we've violated Rule 1, since w(x)1 followed by r(x)0
  So both CPUs cannot be in the critical section.
  In general strict consistency produces intuitive behavior.

How do you implement strict consistency?
  Time:  1  2  3  4
  CPU0:  ST       ST
  CPU1:     LD LD
  Time between instructions << speed-of-light between CPUs!
  How is LD@2 even aware of ST@1?
  How does ST@4 know to pause until LD@3 has finished?
    how does ST@4 know how long to wait?
  Too hard to implement!
```