6.824 2015 Lecture 11: DSM and Sequential Consistency

New Topic: Distributed computing
  The Big Idea: your huge computation on a room full of cheap computers!
  An old and difficult goal; many approaches; much progress; still hot.
  Other cluster computing papers: TreadMarks, MapReduce, Spark
  Sub-topic: sharing framework (RPC? memory? storage? MapReduce?)
  Sub-topic: detailed semantics

Today's approach: distributed shared memory (DSM)
  You all know how to write parallel (threaded) Go programs
  Let's farm the threads out to a big cluster of machines!
  What's missing? shared memory!

DSM plan:
  Programmer writes parallel program: threads, shared variables, locks, &c
  DSM system farms out threads to a cluster of machines
  DSM system creates illusion of single shared memory
  [diagram: LAN, machines w/ RAM, MGR]

DSM advantages
  familiar model -- shared variables, locks, &c
  general purpose (compared to e.g. MapReduce)
  can use existing apps and libraries written for multiprocessors
  lots of machines on a LAN much cheaper than huge multiprocessor

But:
  machines on a LAN don't actually share memory

Approach:
  Use hardware's virtual memory protection (r/w vs r/o vs invalid)
  General idea illustrated with 2 machines:
     Part of the address space starts out on M0
        On M1, marked invalid
     Part of the address space start out on M1
        On M0, marked invalid
  A thread of the application on M1 may refer to an address that lives on M0
    If thread LD/ST to that "shared" address, M1's hardware will take a page fault
       Because page is marked invalid
    OS propagates page fault to DSM runtime
    DSM runtime can fetch page from M0
    DSM on M0, marks page invalid, and sends page to M1
    DSM on M1 receives it from M0, copies it to underlying physical memory
    DSM on M1 marks the page valid
    DSM returns from page fault handler
    Hardware retries LD/ST
  Runs threaded code w/o modification
    e.g. matrix multiply, physical simulation, sort

Challenges:
  Memory model (does memory act like programmers expect it to act?)
  Performance (is it fast?)

How could there be any doubt about how memory acts?

Example
  x and y start out = 0
  thread 0:
    x = 1
    if y == 0:

```
      print yes
  thread 1:
    y = 1
    if x == 0:
      print yes
```

Would it be OK if both threads printed yes?
  Is that even possible?

Could they both print "yes" if this were a Go program?
  How could that happen?
  Why is that allowed?

What is a memory model?
  It explains how program reads/writes in different threads interact.
  A contract:
    It gives the compiler/runtime/hardware some freedom to optimize.
    It gives the programmer some guarantees to rely on.
  You need one on a multiprocessor (e.g. for the labs).
  You *really* need one for a DSM system.
  There are many memory models!
    With different optimization/convenience trade-offs.
    Often called a consistency model.
    Needed for any memory-like or storage system (e.g. the labs).

What does Go's memory model say?
  It answers questions about whether, when and in what order reads and
    writes by different threads interact.
  It answers questions like:
    If a thread says x=1;y=2, could other threads see y=2 but x=0?
    If a thread says x=1;tmp=y, must other threads see x=1 after read completes?
  By default: Go makes NO GUARANTEE about when/whether/order in which one thread's
    memory references interact with other threads.
  A write only becomes visible if writer then interacts with reader
    via a "synchronization event", and then the reader reads.
    E.g. write releases a lock, reader acquires the lock.

So this Go code (from Lab tests!) is not guaranteed to do what I might think:

```
  done := false
  go func() {
    while done == false {
      ...
    }
  }
  ...
  done = true
```

The memory model says the thread may never see the change to done.
  Though in practice, with current compiler, this seems to work.
  Solutions: lock around ALL USES of done; or (better) a channel.

Why does Go break my perfectly intuitive and straightforward code?
  Because an optimizing compiler could thereby generate better code.
  Model allows any optimization that keeps an individual thread's results the same:
    change the order of statements, including variable reads and writes.
      perhaps start a slow read early.
    put a variable in a register, so changes aren't visible to other cores.
    totally eliminate code or variables (done=true is useless!)
    evaluate at compile-time (done is always false!)
  And because some CPU hardware re-orders reads/writes.
    E.g. 2nd write may be visible before 1st if 1st must fetch line from RAM.

E.g. example's load of y may happen before x=1.

Back to DSM.

Naive distributed shared memory
[diagram]
M0, M1, M2, LAN
each machine has a local copy of all of memory
read: from local memory
write: send update msg to each other host (but don't wait)
fast: never waits for communication

Does this naive DSM work well?
What will it do with the example?
This naive DSM is fast but not programmer-friendly.

The paper's memory model: sequential consistency
Formal version of "a read sees the most recent write".
This is a relatively strict and programmer-friendly memory model.
"Most recent" is only meaningful if there's an overall order.

Sequential consistency's definition:
The result of any execution must be the same as if
1. the operations of all the processors executed in some total order,
2. each processor's operations appear in the total order in program order,
3. all operations see results consistent with the total order
i.e. read sees most recent write in the order

Would sequential consistency cause our example to get the intuitive result?
M0: Wx1 Ry?
M1: Wy1 Rx?
The system must get the same results as if it had merged these into one order,
maintaining the order of each machine's operations.
A few possibilities:
Wx1 Ry0 Wy1 Rx1
Wx1 Wy1 Ry1 Rx1
Wx1 Wy1 Rx1 Ry1
and some more symmetric ones (swap M0 and M1)
the second read (either x or y) is always 1, as you'd hope
What is forbidden?
Wx1 Ry0 Wy1 Rx0 -- read didn't see preceding write (naive system did this)
Ry0 Wy1 Rx0 Wx1 -- M0's instructions out of order (some CPUs do this)

This example is a good diagnostic for sequential consistency.

Sequential consistency performance is ok but not great
E.g. system must communicate M0's x=1 to M1 before M0 can proceed to read y
To ensure results consistent with a single total order
(Second "forbidden" example)
This communication takes time!

A simple implementation of sequential consistency
[diagram]
single memory server
each machine sends r/w ops to server, in order, waiting for reply
server picks order among waiting ops
server executes one by one, sending replies

This simple implementation will be slow
single server will get overloaded
no local cache, so all operations wait for server

Which brings us to IVY
  IVY = Integrated shared Virtual memory at Yale
  Memory Coherence in Shared Virtual Memory Systems, Li and Hudak, PODC 1986

IVY big picture
  [diagram: M0 w/ a few pages of mem, M1 w/ a few pages, LAN]
  Operates on pages of memory, stored in machine DRAM (no mem server)
  Each page present in each machine's virtual address space
  On each a machine, a page might be invalid, read-only, or read-write
  Uses VM hardware to intercept reads/writes

Invariant: a page is either:
  Read/write on one machine, invalid on all others; or
  Read/only on >= 1 machines, read/write on none

Basic machinery:
  Read fault on an invalid page:
    Demote R/W (if any) to R/O
    Copy page
    Mark local copy R/O
  Write fault on an R/O page:
    Invalidate all copies
    Mark local copy R/W

IVY allows multiple reader copies between writes
  For speed -- local reads are fast
  No need to force an order for reads that occur between two writes
  Let them occur concurrently -- a copy of the page at each reader

Why crucial to invalidate all copies before write?
  Once a write completes, all subsequent reads *must* see new data
  Otherwise we break our example, and don't get sequential consistency

How does IVY do on the example?
  I.e. could both M0 and M1 print "yes"?
  If M0 sees y == 0,
    M1 hasn't done it's write to y (no stale data == reads see prior writes),
    M1 hasn't read x (each machine in order),
    M1 must see x==1 (no stale data == reads see prior writes).

Message types:
  [don't list these on board, just for reference]
  RQ read query (reader to MGR)
  RF read forward (MGR to owner)
  RD read data (owner to reader)
  RC read confirm (reader to MGR)
  &c

(see ivy-code.txt on web site)

scenario 1: M0 has writeable copy (is owner), M1 wants to read
  [time diagram: MGR 0 1]
  0. page fault on M1, since page must have been marked invalid
  1. M1 sends RQ to MGR
  2. MGR sends RF to M0, MGR adds M1 to copy_set
  3. M0 marks page as access=read, sends RD to M1
  5. M1 marks access=read, sends RC to MGR

scenario 2: now M2 wants to write
  [time diagram: MGR 0 1 2]

```
  0. page fault on M2
  1. M2 sends WQ to MGR
  2. MGR sends IV to copy_set (i.e. M1)
  3. M1 sends IC msg to MGR
  4. MGR sends WF to M0, sets owner=M2, copy_set={}
  5. M0 sends WD to M2, access=none
  6. M2 marks r/w, sends WC to MGR

what if two machines want to write the same page at the same time?

what if one machine reads just as ownership is changing hands?

If M0 issues x=1 at time=10, and M1 issues x=2 at time=20,
  is the resulting x value always x?
  No: messages can be slow, MGR processes them in arbitrary order.
  You cannot use wall-clock time to reason about sequential consistency.

what if there were no IC message?
  (this is The Question)
  i.e. MGR didn't wait for holders of copies to ack?

what if there were no WC message?
  e.g. MGR unlocked after sending WF to M0?
  MGR would send subsequent RF, WF to M2 (new owner)
  What if such a WF/RF arrived at M2 before WD?
    No problem! M2 has ptable[p].lock locked until it gets WD
  RC + info[p].lock prevents RF from being overtaken by a WF
  so it's not clear why WC is needed!
    but I am not confident in this conclusion

In what situations will IVY perform well?
  1. Page read by many machines, written by none
  2. Page written by just one machine at a time, not used at all by others
  Cool that IVY moves pages around in response to changing use patterns

Will page size of e.g. 4096 bytes be good or bad?
  good if spatial locality, i.e. program looks at large blocks of data
  bad if program writes just a few bytes in a page
    subsequent readers copy whole page just to get a few new bytes
  bad if false sharing
    i.e. two unrelated variables on the same page
      and at least one is frequently written
    page will bounce between different machines
      even read-only users of a non-changing variable will get invalidations
    even though those computers never use the same location

What about IVY's performance?
  after all, the point was speedup via parallelism

What's the best we could hope for in terms of performance?
  Nx faster on N machines

What might prevent us from getting Nx speedup?
  Application is inherently non-scalable
    Can't be split into parallel activities
  Application communicates too many bytes
    So network prevents more machines yielding more performance
  Too many small reads/writes to shared pages
    Even if # bytes is small, IVY makes this expensive

How well do they do?
```

```
  Figure 4: near-linear for PDE
  Figure 6: very sub-linear for sort
  Figure 7: near-linear for matrix multiply

Why did sort do poorly?
  "block odd-even merge-split"
  Here's my guess
  N machines, data in 2*N partitions
  Phase 1: Local sort of 2*N partitions for N machines
  Phase 2: 2N-1 merge-splits; each round sends all data over network
  Phase 1 probably gets linear speedup
  Phase 2 probably does not -- limited by LAN speed
    also more machines may mean more rounds
  So for small # machines, local sort dominates, more machines helps
  For large # machines, communication dominates, more machines don't help
  Also, more machines shifts from n*log(n) local sort to n^2 bubble-ish short

How could one speed up IVY?
  next lecture: relax the consistency model
    allow multiple writers to same page!

***

Paper intro says DSM subsumes RPC -- is that true?
  When would DSM be better than RPC?
    More transparent. Easier to program.
  When would RPC be better?
    Isolation. Control over communication. Tolerate latency.
    Portability. Define your own semantics.
  Might you still want RPC in your DSM system? For efficient sleep/wakeup?

Known problems in Section 3.1 pseudo-code
  Fault handlers must wait for owner to send p before confirming to manager
  Deadlock if owner has page r/o and takes write fault
    Worrisome that no clear order ptable[p].lock vs info[p].lock
  Write server / manager must set owner=request_node
  Manager parts of fault handlers don't ask owner for the page
  Does processing of the invalidate request hold ptable[p].lock?
    probably can't -- deadlock

Real-world uses of distributed shared memory?
  (these are from searching the web, not direct knowledge)
  https://numascale.com/numa_technology.html
  http://hazelcast.com/products/hazelcast/
  http://www.dell.com/downloads/global/power/ps1q08-50080247-Intel.pdf
  http://www.scalemp.com/solutions/shared-memory/
```