6.824 2015 Lecture 10: Consistency

Topic: consistency models
  = Interaction of reads/writes on different processors
  Many choices of models!
  Lax model => greater freedom to optimize
  Strict model => matches programmer intuition (e.g. read sees latest write)
  This tradeoff is a huge factor in many designs
  Treadmarks is a case study of relaxing to improve performance

Treadmarks high level goals?
  Better DSM performance
  Run existing parallel code

What specific problems with previous DSM are they trying to fix?
  false sharing: two machines r/w different vars on same page
    M1 writes x, M2 writes y
    M1 writes x, M2 just reads y
    Q: what does IVY do in this situation?
  write amplification: a one byte write turns into a whole-page transfer

First Goal: eliminate write amplification
  don't send whole page, just written bytes

Big idea: write diffs
  on M1 write fault:
    tell other hosts to invalidate but keep hidden copy
    M1 makes hidden copy as well
  on M2 fault:
    M2 asks M1 for recent modifications
    M1 "diffs" current page against hidden copy
    M1 send diffs to M2 (and all machines w/ copy of this page)
    M2 applies diffs to its hidden copy
    M1 marks page r/o

Q: do write diffs change the consistency model?
   At most one writeable copy, so writes are ordered
   No writing while any copy is readable, so no stale reads
   Readable copies are up to date, so no stale reads
   Still sequentially consistent

Q: do write diffs fix false sharing?

Next goal: allow multiple readers+writers
  to cope with false sharing
  => don't invalidate others when a machine writes
  => don't demote writers to r/o when another machine reads
  => multiple *different* copies of a page!
     which should a reader look at?
  diffs help: can merge writes to same page
  but when to send the diffs?
    no invalidations -> no page faults -> what triggers sending diffs?

Big idea: release consistency (RC)
  no-one should read data w/o holding a lock!
    so let's assume a lock server
  send out write diffs on release
    to *all* machines with a copy of the written page(s)

Example 1 (RC and false sharing)

```
x and y are on the same page
M0: a1 for(...) x++ r1
M1: a2 for(...) y++ r2  a1 print x, y r1
What does RC do?
  M0 and M1 both get cached writeable copy of the page
  during release, each computes diffs against original page,
    and sends them to all copies
  M1's a1 causes it to wait until M0's release
    so M1 will see M0's writes

Q: what is the performance benefit of RC?
   What does IVY do with Example 1?
   multiple machines can have copies of a page, even when 1 or more writes
   => no bouncing of pages due to false sharing
   => read copies can co-exist with writers

Q: does RC change the consistency model? yes!
   M1 won't see M0's writes until M0 releases a lock
   I.e. M1 can see a stale copy of x; not possible w/ IVY
   if you always lock:
     locks force order -> no stale reads

Q: what if you don't lock?
   reads can return stale data
   concurrent writes to same var -> trouble

Q: does RC make sense without write diffs?
   probably not: diffs needed to reconcile concurrent writes to same page

Big idea: lazy release consistency (LRC)
  only send write diffs to next acquirer of released lock,
    not to everyone

Example 2 (lazyness)
x and y on same page (otherwise IVY avoids copy too)
everyone starts with a copy of that page
M0: a1 x=1 r1
M1:           a2 y=1 r2
M2:                     a1 print x r1
What does LRC do?
  M2 only asks previous holder of lock 1 for write diffs
  M2 does not see M1's y=1, even tho on same page (so print y would be stale)
What does RC do?
What does IVY do?

Q: what's the performance win from LRC?
   if you don't acquire lock on object, you don't see updates to it
   => if you use just some vars on a page, you don't see writes to others
   => less network traffic

Q: does LRC provide the same consistency model as RC?
   no: LRC hides some writes that RC reveals
   in above example, RC reveals y=1 to M2, LRC does not reveal
   so "M2: print x, y" might print fresh data for RC, stale for LRC
     depends on whether print is before/after M1's release

Q: is LRC a win over IVY if each variable on a separate page?
   or a win over IVY plus write diffs?
   note IVY's fault-driven page reads are lazy at page granularity

Do we think all threaded/locking code will work with LRC?
```

```
  Stale reads unless every shared memory location is locked!
  Do programs lock every shared memory location they read?
  No: people lock to make updates atomic.
      if no concurrent update possible, people don't lock.

Example 3 (programs don't lock all shared data)
x, y, and z on the same page
M0: x := 7 a1 y = &x r1
M1:                    a1 a2 z = y r2 r1
M2:                                      a2 print *z r2
will M2 print 7?
LRC as described so far in this lecture would *not* print 7!
  M2 will see the pointer in z, but will have stale content in x's memory.

For real programs to work, Treadmarks must provide "causal consistency":
  when you see a value,
    you also see other values which might have influenced its computation.
  "influenced" means "processor might have read".

How to track which writes influenced a value?
  Number each machine's releases -- "interval" numbers
  Each machine tracks highest write it has seen from each other machine
    a "Vector Timestamp"
  Tag each release with current VT
  On acquire, tell previous holder your VT
    difference indicates which writes need to be sent
  (annotate previous example)

VTs order writes to same variable by different machines:
M0: a1 x=1 r1  a2 y=9 r2
M1:              a1 x=2 r1
M2:                            a1 a2 z = x + y r2 r1
M2 is going to hear "x=1" from M0, and "x=2" from M1.
  How does M2 know what to do?

Could the VTs for two values of the same variable not be ordered?
M0: a1 x=1 r1
M1:              a2 x=2 r2
M2:                            a1 a2 print x r2 r1

Summary of programmer rules / system guarantees
  1. each shared variable protected by some lock
  2. lock before writing a shared variable
     to order writes to same var
     otherwise "latest value" not well defined
  3. lock before reading a shared variable
     to get the latest version
  4. if no lock for read, guaranteed to see values that
     contributed to the variables you did lock

Example of when LRC might work too hard.
M0: a2 z=99 r2  a1 x=1 r1
M1:                             a1 y=x r1
TreadMarks will send z to M1, because it comes before x=1 in VT order.
  Assuming x and z are on the same page.
  Even if on different pages, M1 must invalidate z's page.
But M1 doesn't use z.
How could a system understand that z isn't needed?
  Require locking of all data you read
  => Relax the causal part of the LRC model
```

Q: could TreadMarks work without using VM page protection?
  it uses VM to
    detect writes to avoid making hidden copies (for diffs) if not needed
    detect reads to pages => know whether to fetch a diff
  neither is really crucial
  so TM doesn't depend on VM as much as IVY does
    IVY used VM faults to decide what data has to be moved, and when
    TM uses acquire()/release() and diffs for that purpose

Performance?

Figure 3 shows mostly good scaling
  is that the same as "good"?
  though apparently Water does lots of locking / sharing

How close are they to best possible performance?
  maybe Figure 5 implies there is only about 20% fat to be cut

Does LRC beat previous DSM schemes?
  they only compare against their own straw-man ERC
    not against best known prior work
  Figure 9 suggests lazyness only a win for Water
    most pages used by most processors, so eager moves a lot of data

What happened to DSM?
  The cluster approach was a great idea
  Targeting *existing* threaded code was not a long-term win
  Overtaken by MapReduce and successors
    MR tolerates faults
    MR guides programmer to good split of data and computation
    BUT people have found MR too rigid for many parallel tasks
  The last word has not been spoken here
    Much recent work on flexible memory-like cluster programming models
    RDDs/Spark, FaRM, Piccolo