Today:

Implementation and example of write-back cache coherence
Limitations of cache coherence
Example uses: Ivy, Coda, Bayou

Difference between linearizability and sequential consistency – is there an implementation that is sequentially consistent, but not linearizable?

Simple case: one processor

Store 1 at x
Add x + 1 -> r1
Store r1 at y
Load from z into r2

In a linearizable system – with an external observer, you would need to do these operations one at a time, wait until each is complete before starting the next.

On a modern single processor CPU, though, these four instructions can execute in parallel – except for the use of x in step 2, and the use of r1 in step 3, everything is independent, provided you can guarantee that z != x or y, you can start the load before the other instructions have finished. So a modern CPU will buffer the write at step 1, do step 2, do step 3 when step 2 is done, and do the load in parallel with steps 1-3, after checking the addresses are different.

But if this is shared memory, life if much much slower. With linearizability, need to wait until the store completes before doing the add. With sequential consistency, can buffer the first store, do it in parallel with the add. The second store must then stall to wait for the first store, but it can still be buffered. But the load must wait for both earlier stores to finish.

And this is bad! Means that it is possible you have two processors, and your system runs several times slower than if it had one processor! Ouch!

Table of: write through/write back vs. coherence: TTL-based, callback

1) write through, TTL (DNS, web)
2) write back, TTL (NFS)
3) write through, coherent (AFS)
4) write back, coherent (coda, ivy)

Illustrate behavior of each quadrant:

TTL – time to live. Allow client to use copy for TTL. After TTL, invalidate the cached copy, so the client goes back to the server to get the latest version.

What semantics does this provide? (eventual)

One tremendous advantage to TTL cache coherence – no state needed at the server. The server does not need to keep track of who has a copy, since they will each time out in turn.

[Anecdote: NFS server. If it crashes, what does it need to do when it reboots? Nothing! Clients can simply retry any RPC's they had in progress, since there is no callback state at the server.]

You can think of NFS as the pesky little sister repeatedly asking: has this item changed? Has this item changed? Until you want to scream, I'll tell you when it does!

That's where callbacks come in. Record state at server as to who has which cached copy, so that server can tell client when its cached data is invalid.

Illustrate state machine for write-through cache coherence: memory can be read-only or invalid on the clients. Server can simply invalidate everyone who has a copy every time there is a change to a value; the clients will go to the server to get the new copy. (Optimization: the server could broadcast the new value.)

Illustrate with two concurrent writes, to two concurrent readers. Readers have item cached. Writers send change through to server; order is the order they reach the server. Server uses callback state to invalidate caches. Then reader has a cache miss, and fetches the value from the server.

Example: AFS: read the file onto the client on file open, write the file back to the server on file close (whole file caching).

Note that this is similar to assignment 1 – the communication with the server is in terms of whole files, but the application running on the client if free to do its operations in terms of normal reads/writes to partial files. The client would then need to turn those into whole file reads and writes.

Can we be more efficient? Write-through means that we must contact the server on every file modification. Imagine the pesky little brother saying: there I changed the file. There I changed it again. And again. Pretty soon you'd say: enough already! Just tell me when you log out!

Write back cache coherence allows for changes to be kept at the client. (Of course, this means that if the client crashes in the meantime, you might lose some of its updates.)

Illustrate state machine for write back cache coherence: owned, read-only, invalid. What causes transition for each state?

This is the algorithm in the Ivy paper. For the project, focus on the basic algorithm -- the Ivy paper discusses a set of further optimizations where the location of the manager can move – this is not important for the project.

Illustrate original example, using write back cache coherence.

```
CPU0:
   v0 = f0();
   done0 = true;
 CPU1:
   while(done0 == false)
    ;
   v1 = f1(v0);
   done1 = true;
 CPU2:
   while(done1 == false)
    ;
   v2 = f2(v0, v1);
```

What cache misses occur, what data is transferred? Initially, nothing cached. Eventually, v1 and v2 gets the right value.

Efficiency of write back versus write through: write back is more efficient if there are repeated writes to the same location; in this example there are no repeated writes. So would write through do the same thing as write back, or does write back have to do more work if there are no repeated writes?

We've been assuming that each variable is independently cached. What happens if some variables share a page? That is called: false sharing. To avoid it, compiler needs to analyze sharing pattern, and put each shared variable on a page with other variables used at the same time.

Here's another example: granularity of sharing. mesh computation. Shared data at the edges between each CPU's region. What if mesh is large, so that each page of data represents a row – then need to transfer the entire mesh on each time step! (Possible solution, used in SVM systems: send only the diffs between versions of the page.)

Earlier I said RPC and cache coherence are duals – move computation to data versus move data to computation. Which is the more efficient way to implement the program above?

Other questions:

**What state do we need to keep for the various levels of coherence? TTL: none!

***Scalability of directory information: to be linearizable, need to a central directory, with a bit per processor, or a list of processors caching each block. (why there isn't cache coherence of web pages.) We could try to scale by distributing the callback state as a tree of callbacks.

***We mentioned that NFS allows for transparent recovery when server or clients fail, using idempotent operations and TTL based coherence. Can we get transparent recovery with callbacks? (Recover by asking others what server state was.)

****Why use write back coherence vs. just write through? (If data is written repeatedly.)

***Why not always use write back coherence? In the presence of failures, would be a blocking protocol, unless you log writes to (multiple disks) to allow remote recovery.

Some examples of cache coherence in practice.

Let's look at IVY. What consistency does it provide and how?

Why is Ivy cool?
All the advantages of *very* expensive parallel hardware.
On cheap network of workstations.
No h/w modifications required!

Do we want a single address space?

Or have programmer understand remote references?
Shall we make a fixed partition of address space?
I.e. first megabyte on host 0, second megabyte on host 1, &c?
And send all reads/writes to the correct host?

Ivy: We can detect reads and writes using VM hardware.
I.e. read- or write-protect remote pages.

What if we didn't do a good job of placing the pages on hosts?
Maybe we cannot predict which hosts will use which pages?

Could move the page each time it is used.
When I read or write, I find current owner, and I take the page.
So need a more dynamic way to find current location of the page.

What if lots of people read a page?

Move page for writing, but allow read-only copies.
When I write a page, I invalidate r/o cached copies.
When I read a non-cached page, I find most recent writer.
Works if pages are r/o and shared, or r/w by one host.

Assume that each variable is on its own virtual memory page. What happens if they are both on the same page?

A few more details about Ivy:
Message types:
RQ read query (reader to MGR)
RF read forward (MGR to owner)
RD read data (owner to reader)
RC read confirm (reader to MGR)
WQ
IV
IC
WF
WD
WC

problem:
a write has many steps and modifies multiple tables
the tables have invariants:
MGR must agree w/ CPUs about the single owner
MGR must agree w/ CPUs about the copy_set
copy_set != {} must agree with owner's writeability
thus write implementation should be atomic

what enforces the atomicity?
what are the RC and WC messages for?
what if RF is overtaken by WF?

does Ivy provide linearizability or sequential consistency?

Invariants:
1. every page has exactly one current owner
2. current owner has a copy of the page
3. if mapped r/w by owner, no other copies
4. if mapped r/o by owner, maybe identical other r/o copies
5. manager knows about all copies

Ivy does seem to use our two seq consist implementation rules. You can construct a total order always:

1. Each CPU to execute reads/writes in program order, one at a time
2. All writes are in a total order (manager orders them)
3. Once read observes effect on write, it is partially ordered behind it. Order the reads in any total order that obeys the partial order.

If you study the protocol carefully, then it is possible to construct an argument that there is no partial order created by the protocol than cannot be embedded in a total order. All CPUs observe all local ops in a local total order (1). All CPUs observe other CPU's operations in order that is consistent with a total order. For writes that is easy to see because they form a total order because there is only a single writer (2). For reads the argument is more complex because reads can happen truely concurrent, but it is never the case that a read on one processor observes a result that is inconsistent with an observation on another processor in a total order. This could only happen if a scenario like A or B above is possible, and the confirmation messages+locks ensure that never happens (3).

Coda: What about disconnected operation?

a) prefetch things you will need in future
b) write back when connected
c) conflict resolution?

Coda conflict resolution policy: changes appear in processor order, applied at time when notebook reconnects.

Is that sequentially consistent? Serializable?

A few questions:

How would you implement google docs or windows live today?

Assume perfect connectivity?

If we wanted to allow offline work?

Seems like you would sync the entire directory, and not rely on automated hoarding. Would an explicit check in/check out model work better?

Email sync: takes minutes, even fully connected.

Bayou: p2p disconnected operation, application-level

Idea is that you should be able to sync a set of portables, without access to a server.

Exchange updates with neighbor; not committed until everyone sees it (and you know that no other earlier updates can occur).