Lecture 5: Cache coherence

Topics:
Memory consistency models
Implementations of memory consistency

Last week: we outlined a few problems with client/server model of computing: performance, fault tolerance, scalability, software engineering, security. We'll deal with the performance this week and most of next week, and that's the focus of assignment 2. Next Friday we'll shift gears and consider failures, and then on through the rest of the topics.

For now, let's assume no one fails, as that makes the logic easier to understand.

If you have an expensive operation, like an RPC to fetch a file, key is to avoid it where possible – that is, to use a local cache to avoid going to the server. The fastest RPC is one you don't make.

Examples of distributed systems that do caching: (pretty much every distributed system!)
Web
Email
cvs
Ipod sync
Distributed file systems: Many clients, one server
DNS (Internet naming)
Shared virtual memory
Multicore architectures
Distributed databases
ORCA (add money, buses aren't updated instantly)

One way to view this: caching is the inverse of an RPC. With RPC, we send the computation to the data; with caching, we bring the data to the computation. If we always send the computation to the data, then the result is simple, if a bit inefficient. (How inefficient – possibly 100K x!) Caching provides an extra dimension of flexibility to the design – location independence for where we put the data and where we put the computation (indeed, we can move around the location depending on the needs of the application). Of course there are issues with security, fault tolerance, etc, that we'll punt for now.

All caching systems face the same set of design issues:
what items to cache (data?, results of computation?)
what to evict (if not enough space to store everything)
where to look on a miss? other clients? Server? What's relative cost of LAN, WAN, disk?
what happens when there is an update? Multiple copies of state that might be stale.

Our focus today: this last question; see 451 for the first two.

Need to start a bit abstractly. Consider a memory, with ability to load/store to memory. We can think of that memory as files, as objects in an RMI system, as DRAM, as disk in a network attached disk. Same issue in each case.

Example 1:

```
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

Problem A:
CPU0's writes of v0 and done0 may be interchanged by network
leaving v0 unset but done0=true

(Q: suppose we implement the memory by sending updates to every node. Would your RPC design have this problem? That is, can a later RPC be processed before an earlier one?)

Can fix this if each CPU sees each other's writes in issue order. Not always a good assumption. For performance, most CPUs issue multiple read/write operations in parallel, without waiting for the previous one to complete (hide high latency operations). For writes, issue them in order, but if they go to different memory banks, may complete (be visible to other processors) in a different order.

If you program in a high level language, the compiler might think each CPU operates independently, on its own local memory, and therefore it is free to say that since v0 and done0 have no data dependency, so that they can be reordered by the compiler.

Problem B:
CPU2 may see CPU1's writes before CPU0's writes
i.e. CPU2 and CPU1 disagree on order of CPU0 and CPU1 writes

(Q: again, assume we implement caching by sending the update to every node. Would your RPC design have this problem?)

The behavior of a program can depend on the "memory model" – that is, what behavior can we expect from memory?

Complexity of the memory model arises out of the desire to do things quickly – if we are content with slow and safe, things can be pretty straightforward.

Simplest model (called single copy, or linearizable memory): the behavior of a set of caches should appear exactly the same as if every process was running on one node with a single memory. In other words, reads and writes should be done in the order they were made in real time – if one processor wrote a location, and another processor later on (in physical time) read the location, then the later one would always see the write.

Sidebar: the nomenclature is highly confusing. In plain English usage, one would think linearizability and serializability mean the same thing, but in terms of cache coherence, they mean different things.

Sidebar: Another bit of subtlety that can trip you up: we think of read and write being instantaneous, but they do take time to perform, eg., because some might need to be sent to a server. Even on a single node reads and writes take time. So:

Start of write

Write finishes

Start of read on a different processor

Value returned

Suppose the start of the read might overlap the time of the write; what value should be returned? Ditto if there are two overlapping writes on A and B – what should be returned by a later read by C? Can a different value be returned to C and D?

In the simple (single copy) model, we only know that the read or write was done somewhere in the window, but not where in the window. Means the system can't finish an operation until all copies have been updated.

Another sidebar:

External vs. internal consistency. Suppose instead of signaling with done0/done1, we use the telephone – I start do some edits on one computer, and when it finishes, I assume that the write is done. So I nudge you on the next computer for you to start

your compilation, using my edits. In that case, there's external synchronization – a causal relationship between events, that is not visible to the system.

[Warning: NFS does not support single copy! So if you do this with your lab partner, it isn't guaranteed to work!]

If we assume there is no external communication, and that we know all the memory references in a system, we can allow a bit more flexibility.

Sequential consistency, or serializability: every operation appears in the order that each processor issued it, and every operation appears on every processor in the same order, but the order may differ from the order they were done in real time.

Specifically, this allows potentially conflicting operations to be done in parallel, provided we make sure that every node sees the same result. Operations appear as if they happened in some single sequential order.

What's an example of the difference between sequential consistency and single copy?

CPU1: write a value

CPU2: write a value (a bit later)

(synchronize CPU3 and CPU4 to ensure the following comes after both writes complete)

CPU3: read value
CPU4: read value

What value does CPU3 and CPU 4 read?

With single copy/linearizability, CPU3 and CPU4 read the same value, the value written by CPU2. With sequential consistency/serializability, we need to make sure that if there are two writes by the same CPU, they appear to other CPUs in order, but it allows CPU1 and CPU2's writes to be in any order, provided every CPU sees them in the same order.

So CPU3 and CPU4 can see either CPU1's write or CPU2's write, but both will see the same one.

[Seems strange, perhaps, but if the programmer intended it to matter what happened, then they'd need to have inserted synchronization between CPU1's write and CPU2's write. Since they occur concurrently, it can't really matter which comes first!]

This is called a "data race" – two memory operations, on different processors, with no intervening synchronization. You could argue this is a bug!

Sequential consistency allows for certain types of compiler and hardware optimizations, especially when there are multiple physical memories. In the absence of external synchronization, e.g., an observer operating in real time, you can't tell the difference – the system is internally consistent, and behaves like there is a single copy of memory. So the program we started with – that will work just fine on a sequentially consistent system!

Third model (causal ordering): a read returns a causally consistent recent version of the data. That is, if I have received a message A from a node (or indirectly received it through some other node), then I will see all updates that node made prior to A.

This relaxes ordering constraints even more, admitting a faster implementation, although at the cost of making the system more complex to reason about for the programmer.

(Question for the audience: is it possible for causal order to differ from sequential consistency?)

In the above example, it would be possible for CPU3 to see the writes as CPU1 then CPU2, while CPU4 can see them in the other order, CPU2 then CPU1. This is not sequentially consistent – not consistent with some sequential order of operations. But it is causally consistent!

So no advantage to doing causal ordering? In most cases, it does what you want. For example, the code we started with, should work correctly with causal ordered memory.

Other forms of weak memory consistency: provide sequential consistency only across explicit synchronization operations. For example, the compiler could reorder memory operations, except that if the system called an explicit "barrier" operation across all the CPUs, then the memory operations before the barrier would need to complete before any of the memory operations after the barrier. If we had a barrier after "done" in the example, program would work correctly.

Another version of this: group operations that need to be sequentially consistent with each other; allow operations in other groups to be done in any order with respect to each other.

Finally: in the limit, eventual consistency: a read returns a recent version of the data, but not necessarily the most recent version, or one that is consistent with any other CPU's read. But if nothing changes for long enough, all CPU's see the change.

Examples: NFS, DNS, web. Why not always do sequential consistency or

linearizability? We'll see that it simplifies the implementation tremendously to provide only eventual consistency, especially when there is replicated data. Then you don't need to instantly propagate every update to every replica – you can do that in the background.

Another reason: if nodes can become disconnected, or we want to provide access to data even when the most up to date copy is unavailable. (some would say that need to use eventual consistency for any highly available system. Amazon loses $50M/minute of downtime, so important for it to be always up, even if it is not always consistent.)

Would initial program work with eventual consistency?

Switch gears to implementation techniques for cache coherent memory

Table of: write through/write back vs. coherence: none, TTL-based, callback

1) no caching (very early dfs, novell)
2) write through, TTL (DNS, web)
3) write back, TTL (NFS)
4) write through, coherent (AFS)
5) write back, coherent (coda, ivy)

Illustrate behavior of each quadrant:

Start simple. No caches – what if every RPC goes to server? What semantics does that provide? (linearizability)

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: DNS uses TTL cache coherence, but client checks only when name is used. USGS web site becomes very slow, every time an earthquake hits California, because everyone's TTL will have expired.]

[Another 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.]

How might we implement something stricter? Callbacks (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 broadcast invalidate 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.)

Example: AFS: read the file onto the client on file open, write the file back to the server occurs 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? AFS requires contacting the server on every file modification. Write back cache coherence allows for changes to be kept at the client. (Of course, this means that if the client crashes, you might lose some of its updates.)

Illustrate state machine for write back cache coherence: owned, read-only, invalid This is the algorithm in the Ivy paper.

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. Or to scale: distribute 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.

If there are multiple memory banks (as there often is today), then on a multicore, sequential consistency is difficult to achieve with write through cache consistency,

because it means that you need to wait for a write to complete, before you can move onto the next operation (e.g., involving a different memory bank, and therefore not being ordered with respect to the previous operation). With write back cache consistency, the writes are applied in the local cache, and so it is easier to enforce that operations occur in order and appear to other processors in the same order. But even here, there are issues, e.g., if I write location A, and read location B, I might want to get the cache line for B while I'm waiting for the invalidation of A to complete. But when I apply the write to A, I need to make sure that I still have a copy of B, as I need to make sure I read B after the write to A.

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?
We can detect reads and writes using VM hardware.
I.e. I 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.
Only bad case is write sharing.
When might this arise? False sharing...

Consider what happens in the example we started with; what does each cache have, and what messages does it send to each other computer?

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).