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 (aka shards of a database), may complete (be visible to other processors) in a different order.

[How can we guarantee that a CPU reads another CPU's writes in issue order? One way would be to slow everything down to a crawl: issue one write, wait for it to complete, before issuing the next write. Or put all data on the same server. We'll see that caching allows us to consider some other implementations.]

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

(what if we implement cache coherence by sending the update to every node? Would your RPC design have this problem?)

Thus, 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 (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 (single copy). 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 the words, linearizability and serializability, mean precisely the same thing – that is, they are synonyms in English. But as technical jargon, they mean very slightly different things, and the difference does not have any relationship to the plain English meaning of these two terms.

[Please! when inventing jargon – don't use English words against their plain meaning!]

sidebar:

Serializability vs. linearizability. 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, which is not visible to the system.

A linearizable system is consistent with single copy semantics, even in the presence of external, real-time communication. This means that the system has to do the operations in the same order that they occur.

A serializable (or sequentially consistent) system is consistent with single copy semantics, but only internally – not with respect to any external communication.

This means that a system can do things in an alternative order, in some cases, if an internal observer can't tell the difference.

[Warning: NFS does not support linearizability! So if you call over to your lab partner, "hey, I'm done editing that file", it isn't guaranteed to work! Use git instead.]

Reached here

Recap definitions:

Serializable, sequentially consistent: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Equivalent definition: operates as if there is only one copy of memory, and all memory operations are serialized

Slightly stronger (linearizable): sequentially consistent, and consistent with real time [means: operates with one copy, rather than "as if" there is one copy]

Sidebar: A 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. The result of two overlapping writes is indeterminate – but we know that if the system is linearizable/serializable, it should be consistent with some serial order.

system can't finish a write operation until all copies have been updated. Once the operation has finished: everyone must see the result of the write.

Difference between serializability and linearizability

serializability means 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.

Serializability/sequential consistency allows for certain types of compiler and hardware optimizations, especially when there are multiple physical memories.

Is this optimization ok in a serializable system? A linearizable one?
Write buffering – to buffer or queue a write at the client, and then continue to execute the next instruction/operation while the write is being performed in the background.

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 there is shared memory, life if much much slower.

Adve paper gives two constraints on implementations of serializability:

Don't do the next write until the previous write is completed, where completed means update/invalidate all copies (if item is cached)
All updates to the same location are serialized

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!

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 serializability? They don't differ for the case we started with at the beginning.)

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.

With causal ordering, 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 same value of memory.

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 or sharded 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.) For example, Amazon's revenue is $100K/minute, and customer purchase rates decline dramatically even for small increases in client response time. So for many sites, it is important for it to be always up and responsive, even if data is not always consistent.

Would initial program work with eventual consistency? Could you modify program so that it would 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 some period of time. 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.]

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!

Reached here

Review: Table of write-through, write-back, TTL, callback

TTL can be inefficient, since if data is being updated frequently, need to check server repeatedly even if data hasn't changed.

So: 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. State is per-client/per-memory object. A memory location can be read-only or invalid on each client.

To keep things consistent, when a node updates the server (write-through), server can simply invalidate everyone who has a copy. Delay response to write until all copies have been invalidated. After invalidation, clients will go to the server to get the new copy.

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.

[Note race conditions:

A client might have data invalidated, and go back to the server, before the remaining clients have received the invalidation.

What if we provided the old data? Not even eventually consistent!

What if we provide the new data? Might be ok, but need to ensure that updates appear at each client in the order they were applied at the server – e.g., invalidates must be applied in order. E.g., if update A, B, C – then shouldn't see C' while I can still read A (if some other node can read A' and C) – that's not serializable.

What if we provide the new data to a refetch, but the server is sharded (callback state spread across two or more machines)? Then to keep the clients seeing the updates in order, either the updates need to be (somehow) ordered across shards, or the refetch must wait until all invalidates have been applied.

(Optimization: the server could broadcast the update, but to be serializable, it needs to ensure that everyone sees the updates in the same order.)

Can we be more efficient than write-through? 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, for each client x each memory object.

Constraints: owned by at most one client (at a time)
Read-only at any client => not owned by any client

What causes transition for each state?

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 are used differently, but are in the same unit of sharing (cache line, distributed object, file)? That is called: false sharing.

To avoid it, compiler/programmer needs to analyze sharing pattern, and put each shared variable on a page with other variables used at the same time/in the same way.

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 serializable, need to have 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.

Ivy: Illusion of shared-memory, using virtual memory paging hardware.
Page table marked as invalid, read-only, read-write (owned).
Take page fault to transition between states.

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 cache coherent address space?
Or perhaps we just want programmer/compiler to issue remote references to remote data, where programmer should manage bringing the data back and forth?

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

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 of 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 truly 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).

Examples of systems that provide weaker consistency guarantees.
DNS: multiple replicas, needed for throughput – a huge number of clients reading data from the DNS servers at the same time. E.g., for the root DNS, several hundred replicas spread around the globe. Caching to reduce the load on the server.

How does update work? Update one replica, that replica replies, and in the background it propagates the change to the other replicas.

Implication: may not read your own writes! Update DNS, read the value, and if you happen to connect to a different replica, you'll see the old value!

Bayou paper an attempt to fix this – what constraints might an application need, even if we aren't serializable.

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).