Lecture #36

****
Review -- 1 min
****
coherency – can you tell that memory system is playing tricks by looking at one address?
consistency – can you tell that the memory system is playing tricks by looking at multiple addresses?
Summary -- snooping:

1) Bus makes broadcast possible -→ allows snooping
2) Bus serializes requests → simplifies protocol
3) DA: busses limit scalability
   -- wide busses, split transaction busses, more complex coherency protocols to reduce bus traffic

4) protocol more complicated than simple diagram

****
Outline - 1 min
****
Directory-based coherency

Consistency
achieving acceptable consistency with good performance algorithms

Parallel Case Studies

****
Lecture - 20 min
****

Directory-based coherency

Large-scale systems

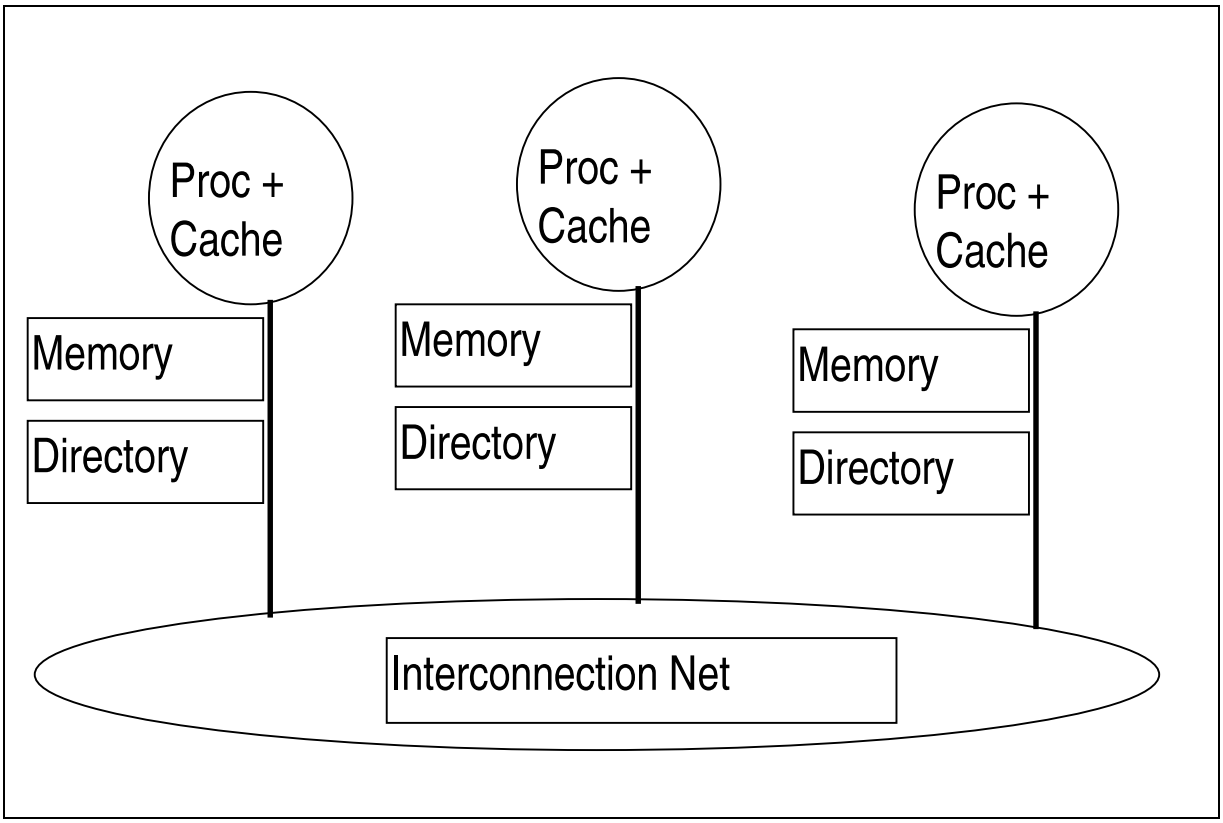
replace bus with scalable interconnect

→ no single place to watch everything go by

Need to check separate data structure

when? On memory accesses

→ put directory by memory



Directory

---

1) Distributed – each directory tracks memory whose "home" is on same node
2) DA: directory scales with size of memory not size of cache
3) distributed – scalable
4) history – centralized directories pre-date snooping

(logical approach – you need to keep track of dependencies → need something like a scoreboard)

Basic Directory Protocol

(Again, simplify – 3 state protocol)

Note – need state machine at cache AND at directory

3 states

1) Shared
2) Uncached
3) Exclusive

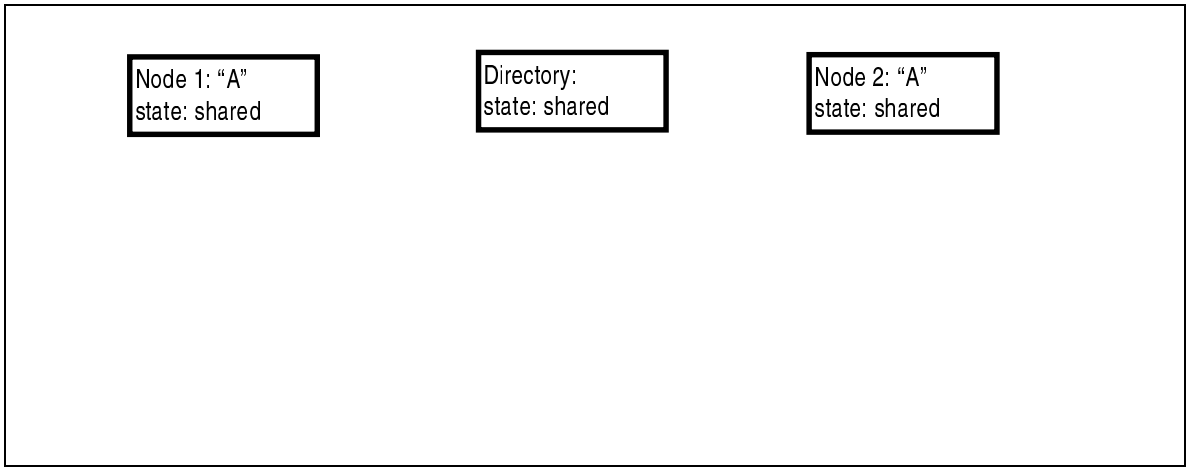
Directory also keeps list of nodes caching the data

- ♦ kept as bit vector (usually)

  (Candidate talk a couple weeks ago – looked at range of ways to keep this data structure)

FIGURE FROM TEXT – cache, directory FSMs

- states identical to snoopy protocol
- transactions v. similar to snoopy
- messages sent to relevant nodes (caching nodes, home node) rather than broadcast on bus



Example: suppose node 1 writes B – what happens?

<Animate on chalk board>

Complications
---

1) Non-atomic operations
   get requests while you are waiting for your request
   to be serviced
      e.g. get invalidate while waiting for upgrade

   solution: cache controller throws away all requests
      that come while waiting for read or write miss

2) Out-of-order messages
   - ♦ network congestion/routing
   - ♦ lost, retransmitted messages

   example – Node 1 read, Node 2 write, directory gets read then
      write, node 1 gets invalidate then data

   example – node 1 has the data and starts write, then gets invalidate
      then gets permission to write
      do I get to keep caching after I write?

   Solution: version numbers, queue or NACK requests, careful
      reasoning about message order

3) avoid deadlock from running out of network buffers
   separate request and reply logical networks (buffer pools actually)
   - ♦ never send a request w/o reserving buffer for reply
   - ♦ allowed to reject request (NAK) but must always drain replies
     from the network
   - ♦ requests can generate replies; replies cannot generate other
     network messages

Side note: same cache consistency protocols for distributed file systems

most systems – centralized directory

xFS – my thesis – includes distributed directory

Tools for Cache Controller Design

Don't underestimate difficulty of the various complications for snooping and directories (and remember, these are still simplified cases)

Design approach

- basically impossible to sit down in a room and write down the correct protocol
- basically impossible to test – bugs come from unexpected ordering of messages – basically unrepeatable
- solution: formal methods, verification tools

/**********

Admin

**********/

/**********

Lecture

**********/

Consistency

Coherency v. Consistency

QUESTION: what is the difference?

Reminder of motivation
Remember definition of coherency:

The problem of Coherency

Time	Event	Cache A	Cache B	Mem
0				1
1	A read X	1		1
2	B read X	1	1	1
3	A write 0 to X	0	1	0
4	B read X	0	1	0

Coherency:

- Any write must eventually be seen by a read
- All writes seen in order

Example (from above) violates first rule – absent coherency via snooping or directories – B could read X==1 indefinitely
Consistency goes to the question of what does "eventually" mean?

> We could definite it in terms of time – "within 1 us" or whatever
> We don't do that

Instead we assume

- communication between processors via memory
- causal model
  > if event X precedes event Y at processor 1, we expect processor 2 to observe that X precedes Y

In our example – if, after P1 updates X, it sets a flag Y saying "I've finished updating X", if P2 reads Y and seems that X has been updated, it had better see the new value of X when it reads X

Simple Consistency Example

P1:
```
A = 0;
…
A = 1;
if(B == 0) {
   printf("P1.");
}
```

P2:
```
B = 0;
…
B = 1;
if(A ==  0){
   printf("P2.");
}
```

Output
"P1."
"P2."
""
"P1.P2."
"P2.P1."

Legal ("Consistent")**?**

Why might this be a problem?
- Write buffer.
- In big MPP – read could go through network faster than write invalidates

Consistency Example

P1:
```
for(ii = 0; ii < 100; ii++) {
    A = ii;
    B = ii;
```

P2:
```
while(1){
    printf("(%d, %d), ",
        A, B);
```

} }

Output: (*Where is inconsistecy*?)

(0,1), (1,1), (1,2), (4,8), (9,9), (9,10), (9,10), (10,10), (11,11), (12,12), …

slide: consistency example

Consistency

notice memory system playing tricks by looking at multiple locations
want: observe updates in consistent/causal order

Consistency – goal: present consistent ('causal') view of memory

Implementing consistency

Analysis of Examples

In first example – two writes by one processor must be observed in same order at another processor
(Fairly obvious definition of causality)

In second and third example – the order between writes and reads must be maintained
(Less obvious?)

→ consistency involves ordering both writes and reads

sequential consistency

reads and writes by a processor are observed in same order that they are executed by a processor

timesharing model – global pattern of reads and writes corresponds to some possible interleaving of sequential processor executions

simple implementation – delay each memory access until the previous one has completed

Problem: write buffers,
lockup free caches,
reads must wait for writes to complete,
can't pipeline memory system, …

→ SLOW

Solution: Weaker consistency models

- allow out-of-order memory accesses (sometimes)
- synchronization operations enforce order when needed

TSO (total store order aka processor consistency)
motivation: processors have write buffers

TSO Consistency model:

- allow reads to bypass writes
- writes complete in order
- write barrier – forces synchronous write flush

Partial store ordering – allows overlap/pipelining of writes
Weak ordering – allow reads and writes to get out of order

Programming Model : Synchronization

We want relaxed consistency models for performance, but how do we avoid confusion like examples.

Think about parallel programming model –

- shared variables protected by locks/monitors
  (even in sequential consistency environment)

- While I hold a lock, no other processor should be reading the things I'm writing anyhow!
- While I don't hold a lock, I should not be reading things that other nodes are writing.

data race free programs

Every write of a variable by one processor is separated from a read/write of that variable by another processor by a pair of synchronization operations – a **release** (unlock) by the first and an **acquire** (lock) by the second.

For TSO, PSO, weak ordering – add acquires and releases that act as read and write fences.

Write fence

- ♦ all writes by P that occur before P executed the write fence complete before the write fence completes

- ♦ no writes by P that occur after the fence are initiated before the fence completes

Read fence similar

Memory fence == both

Release Consistency

TSO, PSO, weak ordering – treat each synchronization as a memory fence

Release consistency distinguishes acquire from release

FIGURE 8.40

Sa → W
Sa → R
R → Sr /* Error in text */
W → Sr

A range of consistency models implemented in hardware:

FIGURE 8.39

Beyond release consistency

Lazy release consistency (figure 8.40)

specific locks w. specific data

Implementation Complexities

Atomic lock operations:

- test and set
- load-linked and store conditional

Efficiency

spin locking
exponential back off
queue locks

****

Summary - 1 min

****

Consistency – want strong semantics, weak performance
Release Consistency