Constructive Computer Architecture

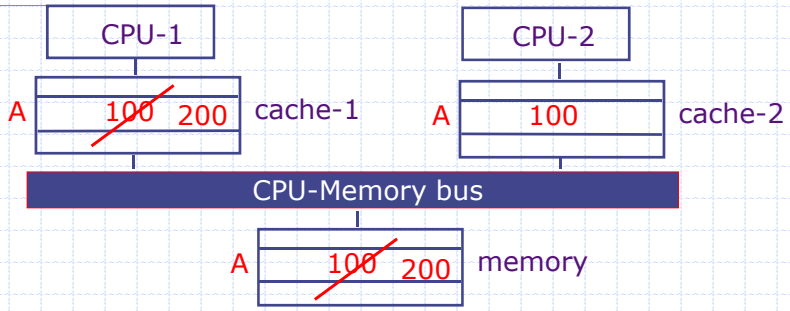
Cache Coherence

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Contributors to the course material

- Arvind, Rishiyur S. Nikhil, Joel Emer, Muralidaran Vijayaraghavan
- Staff and students in 6.375 (Spring 2013), 6.S195 (Fall 2012), 6.S078 (Spring 2012)
  - Asif Khan, Richard Ruhler, Sang Woo Jun, Abhinav Agarwal, Myron King, Kermin Fleming, Ming Liu, Li-Shiuan Peh
- External
  - Prof Amey Karkare & students at IIT Kanpur
  - Prof Jihong Kim & students at Seoul Nation University
  - Prof Derek Chiou, University of Texas at Austin
  - Prof Yoav Etsion & students at Technion

Memory Consistency in SMPs



- Suppose CPU-1 updates A to 200.
  - *write-back:* memory and cache-2 have stale values
  - *write-through:* cache-2 has a stale value

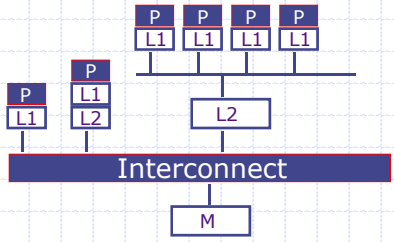
Do these stale values matter?
What is the view of shared memory for programming?

Maintaining Store Atomicity

- *Store atomicity* requires all processors to see writes occur in the same order
  - multiple copies of a location in various caches can cause this to be violated
- To meet the ordering requirement it is sufficient for hardware to ensure:
  - Only one processor at a time has write permission for a location
  - No processor can load a stale copy of the data after a write to the location

⇒ *cache coherence protocols*

A System with Multiple Caches



- Modern systems often have hierarchical caches
- Each cache has exactly one parent but can have zero or more children
- Logically only a parent and its children can communicate directly
- *Inclusion property* is maintained between a parent and its children, i.e.,

  $a \in L_i \Rightarrow a \in L_{i+1}$

Because usually $L_{i+1} >> L_i$

Cache Coherence Protocols

- Write request:
  - the address is *invalidated* in all other caches *before* the write is performed, *or*
  - the address is *updated* in all other caches *after* the write is performed
- Read request:
  - if a dirty copy is found in some cache, that is the value that must be used, e.g., by doing a write-back and reading the memory or forwarding that dirty value directly to the reader.

We will focus on Invalidation protocols as opposed to Update protocols

State needed to maintain Cache Coherence

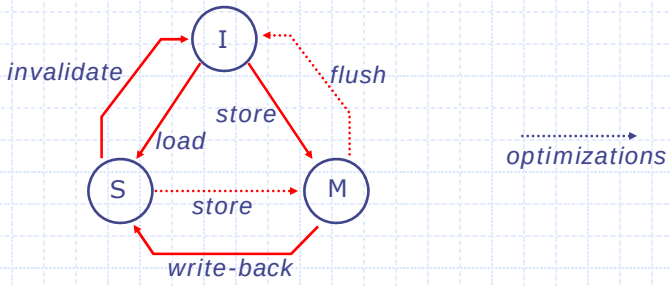
- Use MSI encoding in caches where
  - I *means* this cache does not contain the location
  - S *means* this cache has the location but so may other caches; hence it can only be read
  - M *means* only this cache has the location; hence it can be read and written
- The states M, S, I can be thought of as an order M > S > I
  - A transition from a lower state to a higher state is called an *Upgrade*
  - A transition from a higher state to a lower state is called a *Downgrade*

Sibling invariant and compatibility

- Sibling invariant:
  - Cache is in state M ⇒ its siblings are in state I
  - That is, the sibling states are "compatible"
- The states x, y of two siblings are compatible iff IsCompatible(x, y) is True where

```
IsCompatible(M, M) = False
IsCompatible(M, S) = False
IsCompatible(S, M) = False
All other cases    = True
```

Cache State Transitions



This state diagram is helpful as long as one remembers that each transition involves cooperation of other caches and the main memory

Cache Actions

- On a read miss (i.e., Cache state is I):
  - In case some other cache has the location in state M then write back the dirty data to Memory
  - Read the value from Memory and set the state to S
- On a write miss (i.e., Cache state is I or S):
  - *Invalidate* the location in all other caches and in case some cache has the location in state M then write back the dirty data
  - Read the value from Memory if necessary and set the state to M

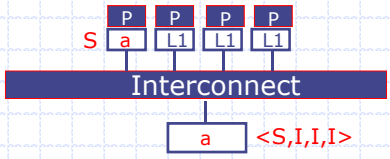
Misses cause Cache upgrade actions which in turn may cause further downgrades or upgrades on other caches

MSI protocol: some issues

- It is possible to have multiple requests for the same location from different processors. Hence there is a need to arbitrate requests
  - In bus-based systems bus controller performs this function
  - In directory-based systems upgrade requests are passed to the parent who acts as an arbitrator
- On a cache miss there is a need to find out the state of other caches
  - In a bus-based system a system-wide broadcast of the request determines the state of other caches by "snooping"
  - In directory-based systems a directory keeps track of the state of each child cache

Directory State Encoding

Two-level (L1, M) system



All addresses in the home memory are in state M

- For each location in a cache, the directory keeps two types of info
  - c.state[a] (*sibling info):* do c's siblings have a copy of location *a*; M (means no), S (means maybe)
  - c.child[c_k][a] (*children info):* the state of c's child c_k for location *a*; At most one child can be in state M
- Since L1 has no children, only sibling information is kept and since main (home) memory has no siblings only children cache information is kept

Directory state encoding *cont*

- New states needed to deal with waiting for responses:
  - c.waitp[a] : Denotes if cache c is waiting for a response from its parent
    - Nothing *means* not waiting
    - Valid (M|S|I) *means* waiting for a response to transition to M or S or I state, respectively
  - c.waitc[c_k][a] : Denotes if cache c is waiting for a response from its child c_k
    - Nothing | Valid (M|S|I)
- Cache state in L1:

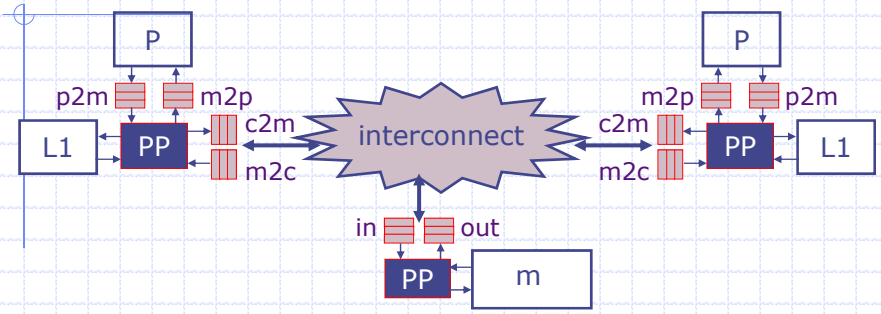
  <(M|S|I), (Nothing | Valid(M|S|I))>
- Directory state in home memory:

  <[(M|S|I), (Nothing | Valid(M|S|I))]>

Children's state

A Directory-based Protocol

an abstract view

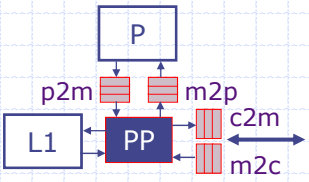


- Each cache has 2 pairs of queues
  - (c2m, m2c) to communicate with the memory
  - (p2m, m2p) to communicate with the processor
- Message format: <cmd, src→dst, a, s, data>

  Req/Resp address state
- FIFO message passing between each (src→dst) pair except a Resp cannot block a Req

Processor Hit Rules

- Load-hit rule
  p2m.msg=(Load a) &
  (c.state[a]>I)
  → p2m.deq;
  m2p.enq(c.data[a]);
- Store-hit rule
  p2m.msg=(Store a v) &
  c.state[a]=M
  → p2m.deq;
  m2p.enq(Ack);
  c.data[a]:=v;



The miss rules are taken care of by the general cache rules to be presented

Processing a Load or a Store miss

Child to Parent: Upgrade-to-y request

Parent to Child: process Upgrade-to-y request

Parent to other child caches: Downgrade-to-x request

Child to Parent: Downgrade-to-x response

Parent waits for all Downgrade-to-x responses

Parent to Child: Upgrade-to-y response

Child receives upgrade-to-y response

Processing a Load miss

- L1 to Parent: Upgrade-to-S request

  (c.state[a]=I) & (c.waitp[a]=Nothing)

  → c.waitp[a]:=Valid S;

  c2m.enq(<Req, c→m, a, S, - >);

- Parent to L1: Upgrade-to-S response

  (∀j, m.waitc[j][a]=Nothing) & c2m.msg=<Req,c→m,a,S,->
  & (∀i≠c, IsCompatible(m.child[i][a],S))

  → m2c.enq(<Resp, m→c, a, S, m.data[a]>);

  m.child[c][a]:=S; c2m.deq

- L1 receiving upgrade-to-S response

  m2c.msg=<Resp, m→c, a, S, data>

  → m2c.deq; c.data[a]:=data; c.state[a]:=S;

  c.waitp[a]:=Nothing;

Processing Load miss *cont.*

What if (∀i≠c, IsCompatible(m.child[i][a],y)) is false?

Downgrade other child caches

- Parent to L1: Upgrade-to-S response

  (∀j, m.waitc[j][a]=Nothing) & c2m.msg=<Req,c→m,a,S,->
  & (∀i≠c, IsCompatible(m.child[i][a],S))

  → m2c.enq(<Resp, m→c, a, S, m.data[a]>);

  m.child[c][a]:=S; c2m.deq

- Parent to Child: Downgrade to S request

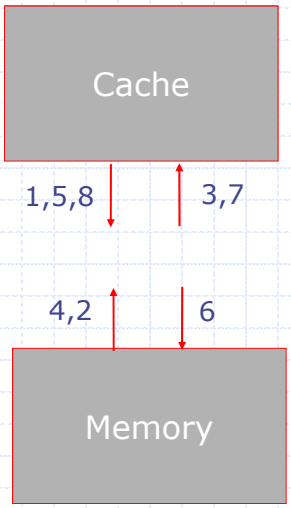
  c2m.msg=<Req,c→m,a,S,-> &
  (m.child[i][a]>S) & (m.waitc[i][a]=Nothing)

  → m.waitc[i][a]:=Valid S; m2c.enq(<Req, m→i, a, S, - >);

Complete set of cache actions

req = {1,4,7}
resp = {2,3,5,6,8}

A protocol specifies cache actions corresponding to each of these 8 different messages



Child Requests

1. Child to Parent: Upgrade-to-y Request
   (c.state[a]<y) & (c.waitp[a]=Nothing)
   → c.waitp[a]:=Valid y;
   c2m.enq(<Req, c→m, a, y, - >);

Parent Responds

2. Parent to Child: Upgrade-to-y response

(∀j, m.waitc[j][a]=Nothing) & c2m.msg=<Req,c→m,a,y,->
& (m.child[c][a]<y) & (∀i≠c, IsCompatible(m.child[i][a],y))
→ m2c.enq(<Resp, m→c, a, y,
(if (m.child[c][a]=I) then m.data[a] else -)>);
m.child[c][a]:=y; c2m.deq;

Child receives Response

3. Child receiving upgrade-to-y response

m2c.msg=<Resp, m→c, a, y, data>
→ m2c.deq;
if(c.state[a]=I) c.data[a]:=data;
c.state[a]:=y;
if(c.waitp[a]=(Valid x) & x≤y) c.waitp[a]:=Nothing;

Parent Requests

4. Parent to Child: Downgrade-to-y Request

(m.child[i][a]>y) & (m.waitc[i][a]=Nothing)

→ m.waitc[i][a]:=Valid y;

m2c.enq(<Req, m→c, a, y, - >);

Child Responds

5. Child to Parent: Downgrade-to-y response

(m2c.msg=<Req,m→c,a,y,->) & (c.state[a]>y)

→ c2m.enq(<Resp, c->m, a, y,

(if (c.state[a]=M) then c.data[a] else -)>);

c.state[a]:=y; m2c.deq

Parent receives Response

6. Parent receiving downgrade-to-y response

c2m.msg=<Resp, c→m, a, y, data>
→ c2m.deq;
if(m.child[c][a]=M) m.data[a]:=data;
c.state[a]:=y;
if(m.waitc[c][a]=(Valid x) & x≥y)
m.waitc[c][a]:=Nothing;

Child receives served Request

7. Child receiving downgrade-to-y request

(m2c.msg=<Req, m→c, a, y, - >) & (c.state[a]≤y)
→ m2c.deq;

Child Voluntarily downgrades

8. Child to Parent: Downgrade-to-y response (vol)

(c.waitp[a]=Nothing) & (c.state[a]>y)

→ c2m.enq(<Resp, c->m, a, y,

(if (c.state[a]=M) then c.data[a] else -)>);

c.state[a]:=y;

Some properties

- Rules 1 to 8 are complete - cover all possibilities and cannot deadlock or violate cache invariants
- Our protocol maintains two important invariants:
  - Directory state is always a conservative estimate of a child's state
  - Every request eventually gets a corresponding response (assuming responses cannot be blocked by requests and a request cannot overtake a response for the same address)
- Starvation, that is a Load or store request is ignored indefinitely has to be prevented; Fair arbitration at the memory between requests from various caches will ensure starvation freedom.

FIFO property of queues

- If FIFO property is not enforced, then the protocol can either deadlock or update with wrong data
- A deadlock scenario:

1. Child 1 requests upgrade (from I) to M (msg1)
2. Parent responds to Child 1 with upgrade from I to M (msg2)
3. Child 2 requests upgrade (from I) to M (msg2)
4. Parent requests Child 1 for downgrade (from M) to I (msg3)
5. msg3 overtakes msg2
6. Child 1 sees request to downgrade to I and drops it
7. Parent never gets a response from Child 1 for downgrade to I

H and L Priority Messages

- At the memory, unprocessed request messages cannot block reply messages. Hence all messages are classified as H or L priority.
  - all messages carrying replies are classified as high priority
- Accomplished by having separate paths for H and L priority
  - In Theory: separate networks
  - In Practice:
    - Separate Queues
    - Shared physical wires for both networks