Constructive Computer Architecture

Cache Coherence

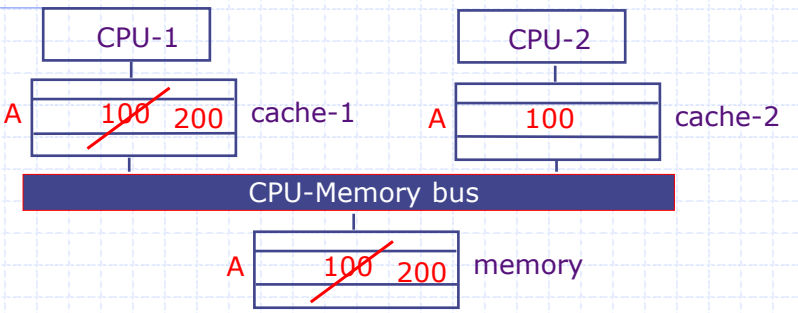
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Includes slides from L21

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

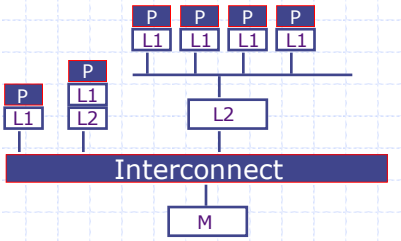
Do these stale values matter?
What is the view of shared memory for programming?

Maintaining Store Atomicity

- *Store atomicity* requires all processors to see writes occur in the same order
  - multiple copies of an address 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 address
  - No processor can load a stale copy of the data after a write to the address

⇒ *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
- Read request:
  - if a dirty copy is found in some cache, that value must be used by doing a write-back and then reading the memory or forwarding that dirty value directly to the reader

State needed to maintain Cache Coherence

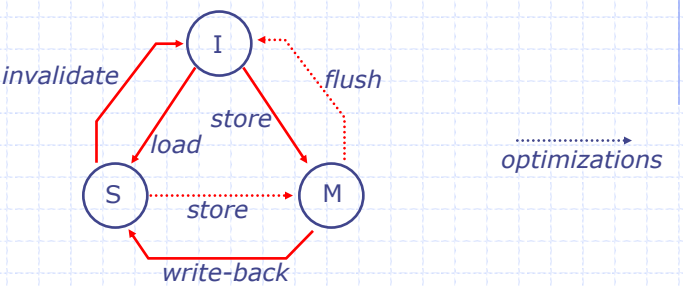
- Use MSI encoding in caches where
  - I *means* this cache does not contain the address
  - S *means* this cache has the address but so may other caches; hence it can only be read
  - M *means* only this cache has the address; hence it can be read and written
- The states M, S, I can be thought of as 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 $\Rightarrow$ its siblings are in state I
  - That is, the sibling states are "compatible"

    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 to maintain the sibling invariants

Cache Actions

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

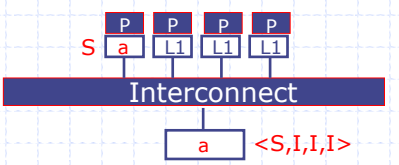
Misses cause Cache upgrade actions which in turn may cause further downgrades or upgrades on other caches

MSI protocol: some issues

- It never makes sense to have two outstanding requests for the same address from the same processor/cache
- It is possible to have multiple requests for the same address from different processors. Hence there is a need to arbitrate requests
- On a cache miss there is a need to find out the state of other caches
- A cache needs to be able to evict an address in order to make room for a different address
  - Voluntary downgrade
- Memory system (higher-level cache) should be able to force a lower-level cache to downgrade

Directory State Encoding

Two-level (L1, M) system



- For each address in a cache, the directory keeps two types of info
  - c.state[a] (*sibling info):* do c's siblings have a copy of address *a*; M (means no), S (means maybe)
  - m.child[c_k][a] (*children info):* the state of child c_k for address *a*; At most one child can be in state M

Directory state encoding

wait states

- 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
  - m.waitc[c_k][a] : Denotes if memory m is waiting for a response from its child c_k
    - Nothing | Valid (M|S|I)
- Cache state in L1:

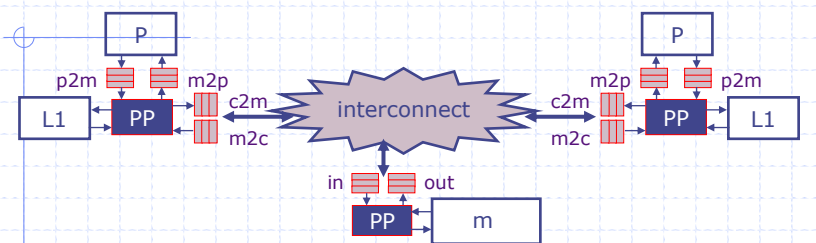
  <(M|S|I), (Nothing | Valid(M|S|I))>
- Directory state in home memory:

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

Children's state

A Directory-based Protocol

an abstract view



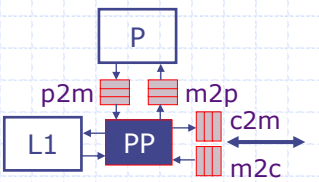
- Each cache has 2 pairs of queues
  - (c2m, m2c) to communicate with the memory
  - (p2m, m2p) to communicate with the processor
- Message format: <cmd, src→dst, a, s, data>

  Req/Resp address state
- FIFO message passing between each (src→dst) pair except a Req cannot block a Resp
- Messages in one src→dst path cannot block messages in another src→dst path

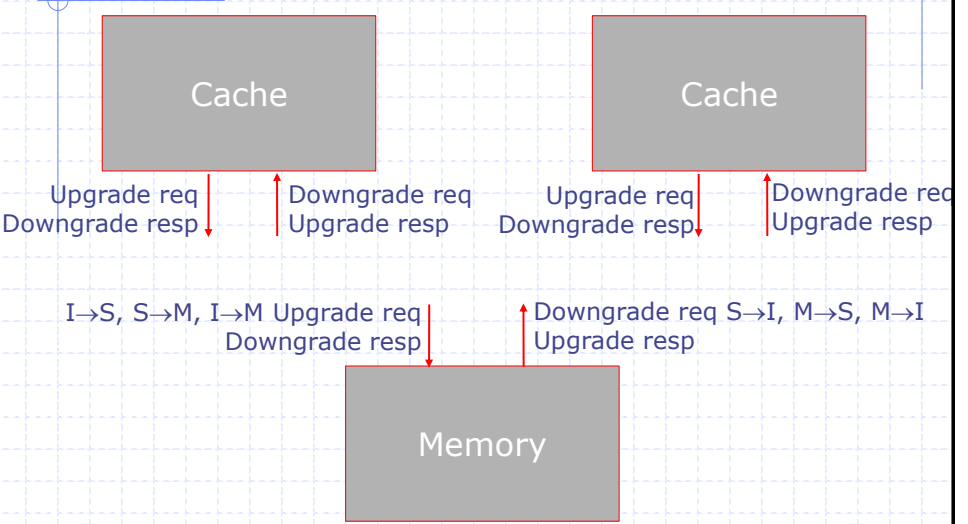
Processor Hit Rules

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

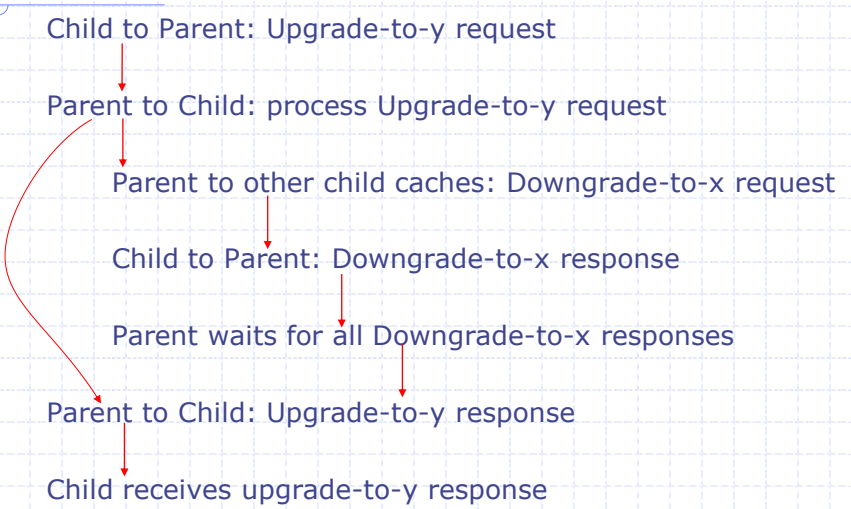
- Store-hit rule
  p2m.msg=(Store a v) &
  c.state[a]=M
  → p2m.deq;
  m2p.enq(Ack);
  c.data[a]:=v;



Processing misses: Requests and Responses



Processing a Load or a Store miss incomplete



Processing a Load miss

ad hoc attempt

- 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, - >);

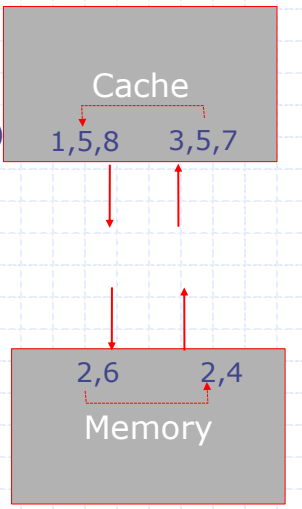
It is difficult to design a protocol in this manner

Invariants for a CC-protocol design

- Directory state is always a conservative estimate of a child's state
  - E.g., if directory thinks that a child cache is in S state then the cache has to be in either I or S state
- For every request there is a corresponding response, though sometimes a response may have been generated even before the request was processed
- Communication system has to ensure that
  - responses cannot be blocked by requests
  - a request cannot overtake a response for the same address
- At every merger point for requests, we will assume fair arbitration to avoid starvation

Complete set of cache/memory actions

1 Up req send (cache)
2 Up req proc, Up resp send (memory)
3 Up resp proc (cache)
4 Dn req send (memory)
5 Dn req proc, Dn resp send (cache)
6 Dn resp proc (memory)
7 Dn req proc, drop (cache)
8 Voluntary Dn resp (cache)



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, - >);

This is a blocking cache since we did not deque the requesting message in case of a miss

Parent Responds

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

(∀j, m.waitc[j][a]=Nothing) & c2m.msg=<Req,c→m,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;
c.waitp[a]:=Nothing;
// the child must be waiting for a state ≤ y

Parent Requests

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

c2m.msg=<Req,c→m,a,y,-> &
(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;
m.child[c][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;

Rules 1 to 8 are complete - cover all possibilities and cannot deadlock or violate cache invariants

Are the rules exhaustive?

Parent rules

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

(∀j, m.waitc[j][a]=Nothing) & c2m.msg=<Req,c→m,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;

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

Rule 4 will get invoked

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

c2m.msg=<Req,c→m,a,y,-> &
(m.child[i][a]>y) & (m.waitc[i][a]=Nothing)
→ m.waitc[i][a]:=Valid y; m2c.enq(<Req, m→i, a, y, - >);

No deq, hence the request is kept pending at the head of the queue;
The address is marked as "busy", i.e., waiting

Are rules exhaustive?

Parent rules

2. Parent to Child: Upgrade-to-y response
   (∀j, m.waitc[j][a]=Nothing) & c2m.msg=<Req,c→m,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;
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, - >);
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;

What if (∀j, m.waitc[j][a]=Nothing) is False?
It is OK not to process the request because this condition will eventually be cleared out

Is every rule necessary?

Consider rule 7 for cache

7. Child receiving downgrade-to-y request
   (m2c.msg=<Req, m→c, a, y, - >) & (c.state[a]≤y)
   → m2c.deq;

A downgrade request comes but the cache is already in the downgraded state

Can happen because of voluntary downgrade

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;

More rules?

- How about a voluntary upgrade rule from parent?

Parent to Child: Upgrade-to-S response (vol)

(m.waitc[c][a]=Nothing) & (m.cstate[c][a]=S)

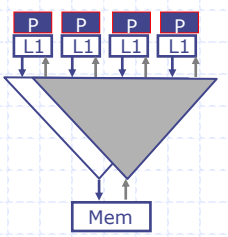
→ m2c.enq(<Resp, m->c, a, M, ->;

m.cstate[c][a]:=M;

The child could have simultaneously evicted the line, in which case the parent eventually makes m.cstate[c][a] = I while the child makes its c.state[a] = M. This breaks our invariant

A cc protocol is like a Swiss watch, even the smallest change can easily (and usually does) introduce bugs

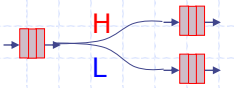
Communication Network



- Two virtual networks:
  - For requests and responses from cache to memory
  - For requests and responses from memory to caches
- Each network has H and L priority messages - a L message can never block an H message other than that messages are delivered in FIFO order

H and L Priority Messages

- At the memory, unprocessed request messages cannot block reply messages.
- H and L messages can share the same wires but must have separate queues



An L message can be processed only if H queue is empty

FIFO property of queues

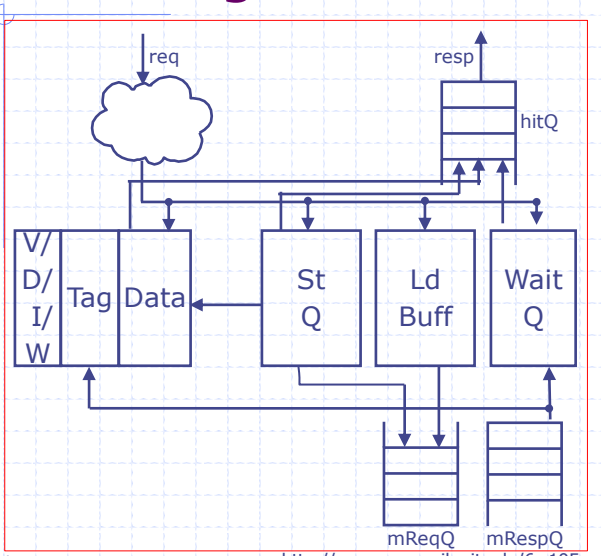
- If FIFO property is not enforced, then the protocol can either deadlock or update with wrong data
- A deadlock scenario:

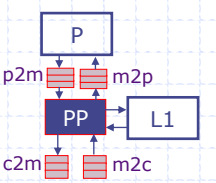
1. Child 1 requests upgrade (from I) to M (msg1)
2. Parent responds to Child 1 with upgrade from I to M (msg2)
3. Child 2 requests upgrade (from I) to M (msg3)
4. Parent requests Child 1 for downgrade (from M) to I (msg4)
5. msg4 overtakes msg2
6. Child 1 sees request to downgrade to I and drops it
7. Parent never gets a response from Child 1 for downgrade to I

Deadlocks due to buffer space

- A cache or memory always accepts a response, thus responses will always drain from the network
- From the children to the parent, two buffers are needed to implement the H-L priority. A child's req can be blocked and generate more requests
- From parent to all the children, just one buffer in the overall network is needed for both requests and responses because a parent's req only generates responses

Integrating PP into a non-blocking cache





Some cache rules need to be changed