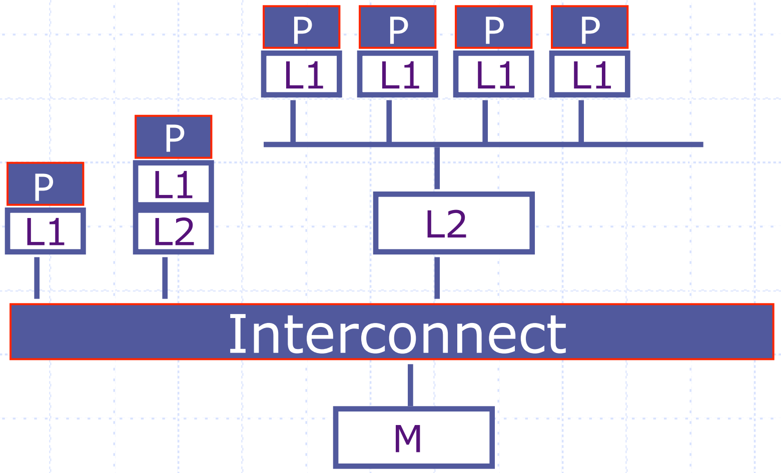
Constructive Computer Architecture

# Cache Coherence

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

# Shared Memory Systems

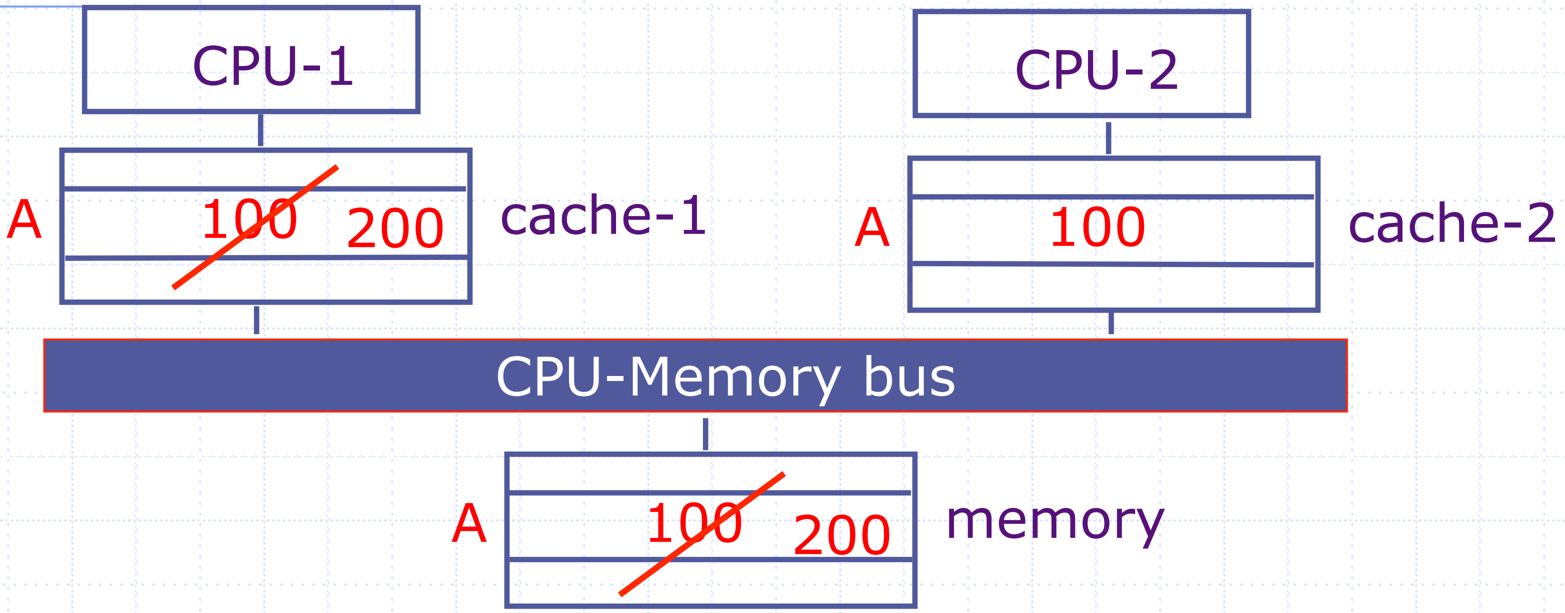


- Modern systems often have hierarchical caches
- Each cache has exactly one parent but can have zero or more children
- Logically only a parent and its children can communicate directly
- *Inclusion property* is maintained between a parent and its children, i.e.,

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

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

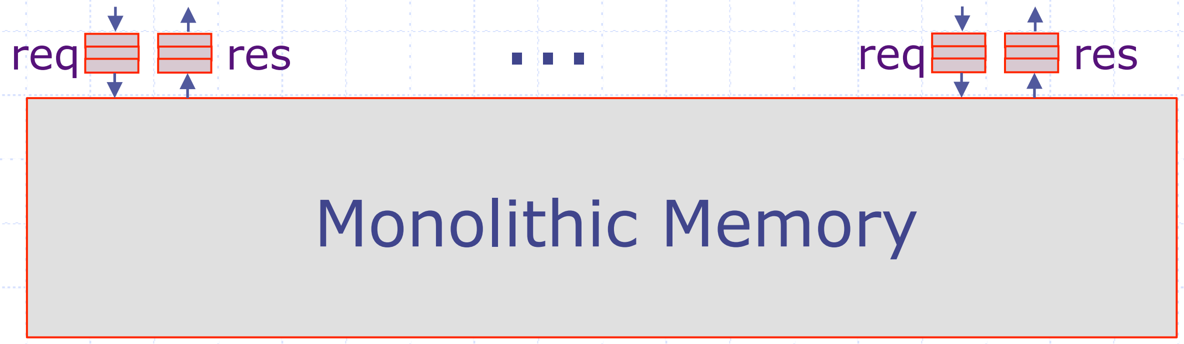
# Cache-coherence problem



- 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?*

## Cache-Coherent Memory



- A monolithic memory processes one request at a time; it can be viewed as processing requests instantaneously
- A memory with hierarchy of caches is said to be *coherent*, if functionally it behaves like the monolithic memory

# Maintaining Coherence

- In a *coherent memory* all loads and stores can be placed in a global order
  - multiple copies of an address in various caches can cause this property to be violated
- This property can be ensured if:
  - Only one cache at a time has the write permission for an address
  - No cache can have a stale copy of the data after a write to the address has been performed

$\Rightarrow$ *cache coherence protocols are used to maintain coherence*

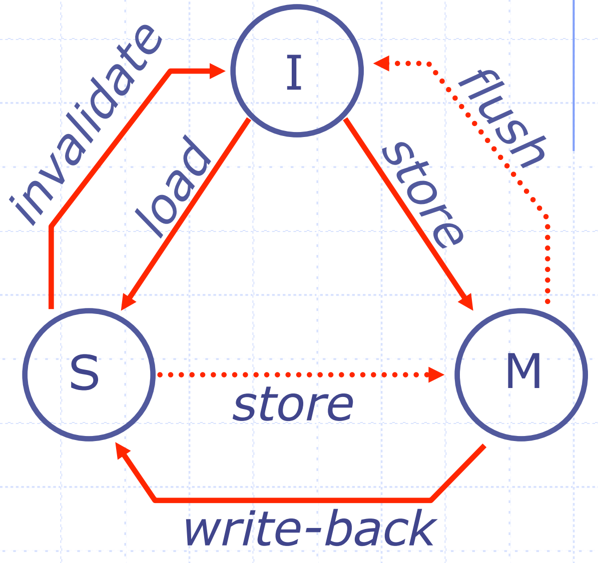
# Cache Coherence Protocols

- 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 then that value is written back to the memory and supplied to the reader. Alternatively the dirty value can be forwarded directly to the reader

*Such protocols are called Invalidation-based*

# State needed to maintain Cache Coherence

- MSI encoding
  - I - cache doesn't contain the address
  - S- cache has the address but so may other caches; hence it can only be read
  - M- 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
  - *Upgrade:* A cache miss causes transition from a lower state to a higher state
  - *Downgrade:* A write-back or invalidation causes a transition from a higher state to a lower state

# Cache Actions

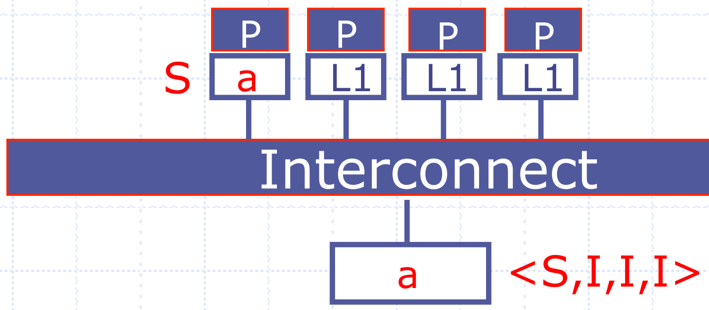
- On a read miss (i.e., Cache state is I):
  - In case some other cache has the address in state M then write back the dirty data to Memory
  - Read the value from Memory and set the state to S
- On a write miss (i.e., Cache state is I or S):
  - *Invalidate* the address in all other caches and in case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

*How do we know the state of other caches?*

Directory

# Directory State Encoding

## Two-level (L1, M) system



- A directory is maintained at each cache to keep track of the state of its children's caches
  - m.child[$c_k$][a]: the state of child $c_k$ for address *a*; At most one child can be in state M
- In case of cache hierarchy > 2, the directory also keeps track of the sibling information
  - c.state[a]: M means c's siblings do not have a copy of address *a*; S means they might

# Directory state encoding

transient states to deal with waiting for responses

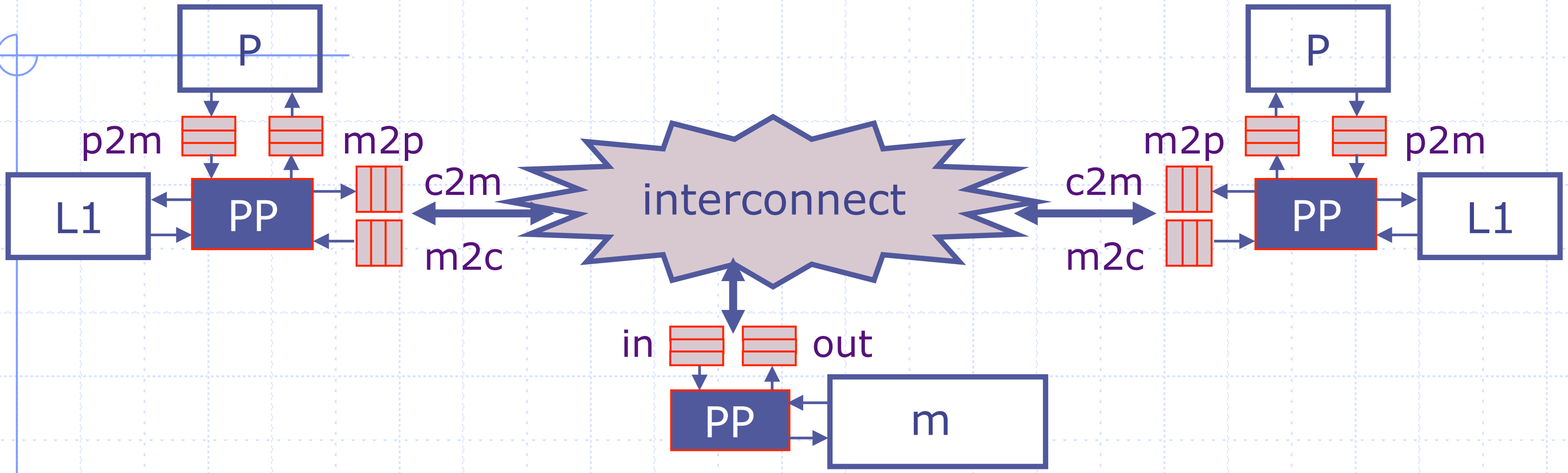
- L1
  - wait state is captured in mshr
- Directory in home memory
  - m.waitc[$c_k$][a] : Denotes if memory m is waiting for a response from its child $c_k$
    - No | Yes
  - <[(M|S|I), (No | Yes)]>

Child's state

Waiting for downgrade response

# A Directory-based Protocol

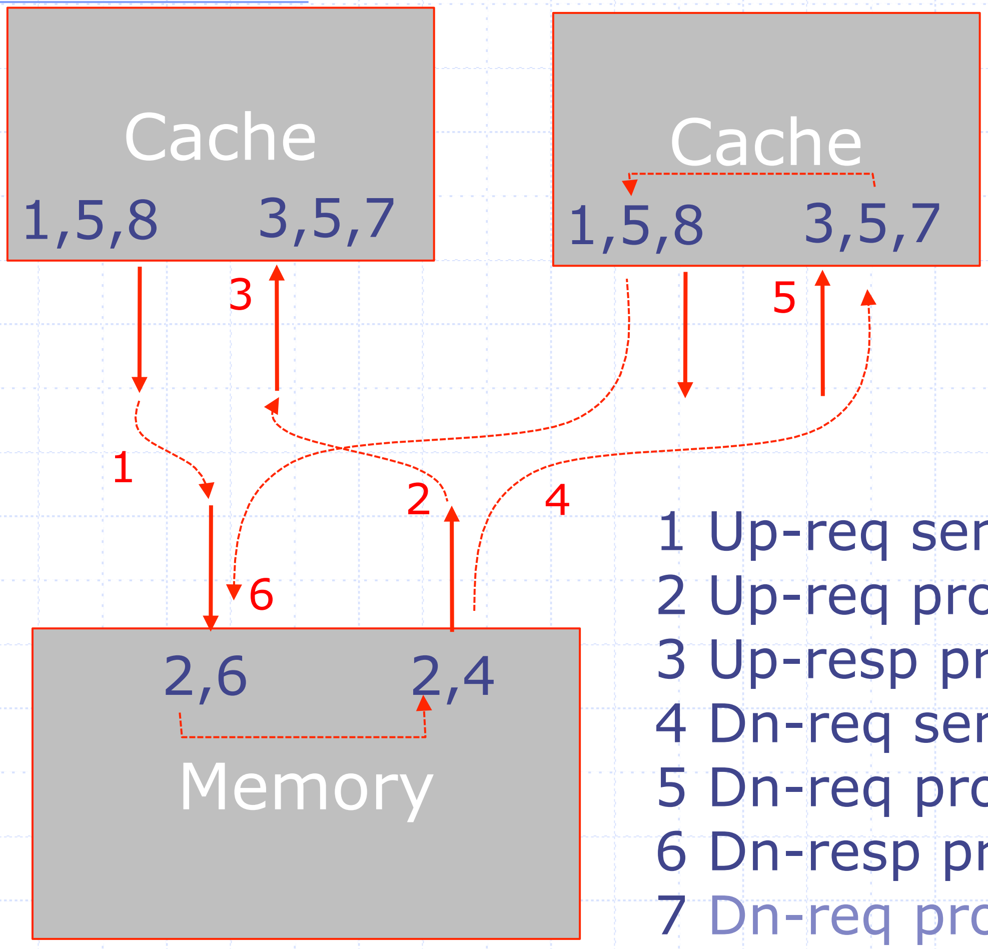
*an abstract view*



- 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

# Processing misses: Requests and Responses



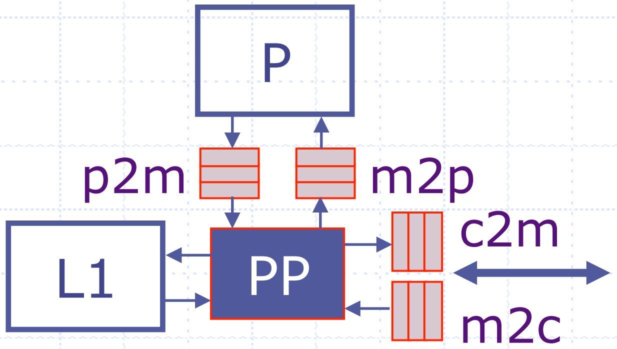
1 Up-req send (cache)
2 Up-req proc, Up resp send (memory)
3 Up-resp proc (cache)
4 Dn-req send (memory)
5 Dn-req proc, Dn resp send (cache)
6 Dn-resp proc (memory)
7 Dn-req proc, drop (cache)
8 Voluntary Dn-resp (cache)

## CC protocol for blocking caches

Extension to the Blocking L1 design discussed in L17-18

# Req method
## hit processing

```
method Action req(MemReq r) if(mshr == Ready);
  let a = r.addr;
  let hit = contains(state, a);
  if(hit) begin
    let slot = getSlot(state, a);
    let x = dataArray[slot];
    if(r.op == Ld) hitQ.enq(x);
    else // it is store
          if (isStateM(state[slot])
               dataArray[slot] <= r.data;
          else begin missReq <= r; mshr <= SendFillReq;
                     missSlot <= slot; end
           end
  else begin missReq <= r; mshr <= StartMiss; end // (1)
endmethod
```



# Start-miss and Send-fill rules

```
Rdy -> StrtMiss -> SndFillReq -> WaitFillResp -> Resp -> Rdy
```

```
rule startMiss(mshr == StartMiss);
  let slot = findVictimSlot(state);
  if(!isStateI(state[slot]))
    begin // write-back (Evacuate)
      let a = getAddr(state[slot]);
      let d = (isStateM(state[slot])? dataArray[slot]: -);
      state[slot] <= (I, _);
      c2m.enq(<Resp, c->m, a, I, d>); end
  mshr <= SendFillReq; missSlot <= slot; endrule

 rule sendFillReq (mshr == SendFillReq);
   let upg = (missReq.op == Ld)? S : M;
   c2m.enq(<Req, c->m, missReq.addr, upg, - >);
   mshr <= WaitFillResp;  endrule  // (1)
```

# Wait-fill rule and Proc Resp rule

```
Rdy -> StrtMiss -> SndFillReq -> WaitFillResp -> Resp -> Rdy
```

```
rule waitFillResp(mshr == WaitFillResp);
  let <Resp, m->c, a, cs, d> = m2c.msg;
  let slot = missSlot;
  dataArray[slot] <=
        (missReq.op == Ld)? d : missReq.data;
  state[slot] <= (cs, a);
  m2c.deq;
  mshr <= Resp;
endrule // (3)
```

```
rule sendProc(mshr == Resp);
  if(missReq.op == Ld) begin
    c2p.enq(dataArray[slot]); end
  mshr <= Ready;
endrule
```

## Parent Responds

```
rule parentResp;
   let <Req,c->m,a,y,-> = c2m.msg;
   if((∀i≠c, isCompatible(m.child[i][a],y))
      && (m.waitc[c][a]=No)) begin
     let d = ((m.child[c][a]=I)? m.data[a]: -);
     m2c.enq(<Resp, m->c, a, y, d);
     m.child[c][a]:=y;
     c2m.deq;
   end
endrule
```

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

## Parent (Downgrade) Requests

```
rule dwn;
  let <Req,c->m,a,y,-> = c2m.msg;
  if (!isCompatible(m.child[i][a], y) &&
         (m.waitc[i][a]=No))
  begin
    m.waitc[i][a] <= Yes;
    m2c.enq(<Req, m->i, a, (y==M?I:S), - >);
  end
Endrule // (4)
```

This rule will execute as long some child cache is not compatible with the incoming request

## Parent receives Response

```
rule dwnRsp;
  let <Resp, c->m, a, y, data> = c2m.msg;
  c2m.deq;
  if(m.child[c][a]=M) m.data[a]<=data;
  m.child[c][a]<=y;
  m.waitc[c][a]<=No;
endrule // (6)
```

## Child Responds

```
rule dng(mshr != Resp);
  let <Req,m→c,a,y,-> = m2c.msg;
  let slot = getSlot(state,a);
  if(getCacheState(state[slot])>y) begin
    let d = (isStateM(state[slot])? dataArray[slot]: -);
    c2m.enq(<Resp, c->m, a, y, d>);
    state[slot] := (y,a);
  end
  // the address has already been downgraded
  m2c.deq;
endrule // (5) and (7)
```

## Child Voluntarily downgrades

```
rule startMiss(mshr == Ready);
  let slot = findVictimSlot(state);
  if(!isStateI(state[slot]))
    begin // write-back (Evacuate)
      let a = getAddr(state[slot]);
      let d = (isStateM(state[slot])? dataArray[slot]: -);
      state[slot] <= (I, _);
      c2m.enq(<Resp, c->m, a, I, d>);
   end
endrule // (8)
```

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

## 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 it is generated even before the request is 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

## 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
- 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
  - caches need to keep track of the state of their children’s caches