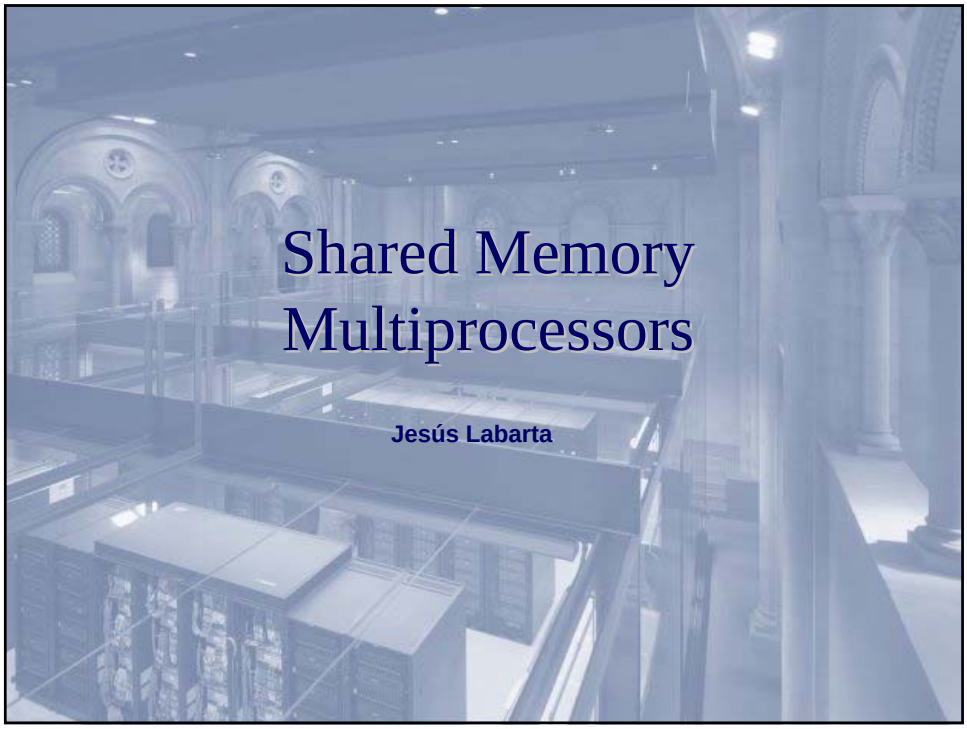
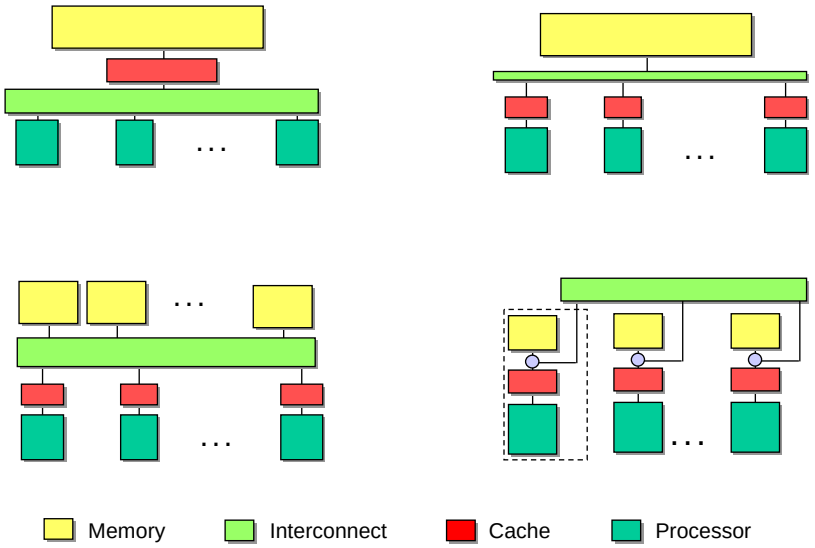


Shared Memory Multiprocessors

Jesús Labarta

Index

Shared Memory architectures



Concepts ?

- **Memory**
- **Time**

Concepts ?

- **Memory**
  - Load/store (@)
  - Containers
- **Time**
  - Order
    - ✓ happens “before than”
    - ✓ “last”
  - Interval
    - ✓ How many things I can do between two events
- **Relativity**
  - Metrics/relations defined for a specific **observer**

Concepts ?

- **Time in multiprocessor system**
  - Local time
    - ✓ Program order
    - ✓ Interval between instructions
      - – Elastic: drift, preemptions
  - Global time
    - ✓ Observable by all processors ?
    - ✓ Propagation of signals
  - Mapping of local time to global time
    - ✓ Problem: Ordering of events when timescales comparable to propagation of signal, drift,… ?

Concepts ?

- **Memory operation**
  - Issued: presented to the memory subsystem
  - Performed: effect achieved
  - Complete: performed with respect to all processors

Issues

- **Coherence**
- **Consistency**

Coherence

- **Every read sees the “last” write**
  - Order
    - ✓ Clear if long distance between events
    - ✓ Fuzzy in the short distances
  - Who defines “last” ?

Coherence

- **The problem**
  - u=5

```
   P1            P2            P3
 ...           ...           ...
Load r1,u     Load r1,u     Load r1,u
 ...           ...           ...
Load r2,u     Load r2,u     Store 7,u
r2=r2+3       ...           ...
Store r2,u    Load r3,u     Load r2,u
...
Load r3,u

 r1= ?         r1= ?         r1= ?
 r2= ?         r2= ?         r2= ?
 r3= ?         r3= ?
```

Coherence

- u=5

Local / Absolute time /Global order ???

```
   P1            P2            P3
 ...           ...           ...
Load r1,u     Load r1,u     Load r1,u
              ...
 ...          Load r2,u
                            Store 7,u
Load r2,u     ...           ...
r4=r2+3
Store r4,u    Load r3,u     Load r2,u
...
Load r3,u
```

```
P1: Load r1,u
P3: Load r1,u
P2: Load r1,u
P2: Load r2,u
P3: Store 7,u
P1: Load r2,u
P1: Store r4,u
P2: Load r3,u
P3: Load r2,u
P1: Load r3,u
```

```
r1= 5         r1= 5         r1= 5
r2= 7         r2= 5         r2= 10
r3= 10        r3= 10
```

Coherent memory behavior

Coherence

- u=5

```
   P1            P2            P3
 ...           ...           ...
Load r1,u     Load r1,u     Load r1,u
                            ...
 ...          ...           Store 7,u
              Load r2,u
Load r2,u     ...           ...
r4=r2+3
Store r4,u    Load r3,u
...                         Load r2,u
Load r3,u
```

```
P1: Load r1,u
P3: Load r1,u
P2: Load r1,u
P3: Store 7,u
P2: Load r2,u
P1: Load r2,u
P1: Store r4,u
P2: Load r3,u
P3: Load r2,u
P1: Load r3,u
```

```
r1= 5         r1= 5         r1= 5
r2= 7         r2= 7         r2= 10
r3= 10        r3= 10
```

Coherent memory behavior

Coherence

- u=5

```
   P1            P2            P3
 ...           ...           ...
Load r1,u     Load r1,u     Load r1,u
                            ...
                            Store 7,u
 ...           ...
                            ...
Load r2,u
r4=r2+3       Load r2,u     Load r2,u
Store r4,u
...           ...
Load r3,u     Load r3,u
```

```
 r1= 5         r1= 5         r1= 5
 r2= 5         r2= 5         r2= 7
 r3= 8         r3= 5
```

```
P1: Load r1,u
P3: Load r1,u
P2: Load r1,u
P1: Load r2,u
P2: Load r2,u
P2: Load r3,u
P3: Store 7,u
P3: Load r2,u
P1: Store r4,u
P1: Load r3,u
```

Coherent memory behavior ??

Coherence

- U=5

```
   P1            P2            P3
 ...           ...           ...
Load r1,u     Load r1,u     Load r1,u
 ...                         ...
Load r2,u                   Store 7,u
r4=r2+3       ...
Store r4,u                  ...
...           Load r2,u     Load r2,u
Load r3,u
              ...
              Load r3,u
```

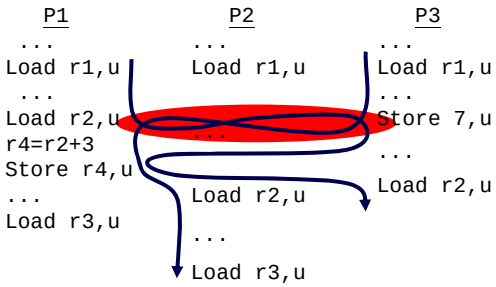
```
 r1= 5         r1= 5         r1= 5
 r2= 7         r2= 10        r2= 7
 r3= 10        r3= 7
```

Non Coherent memory behavior

How could this happen ??

Coherence

- U=5

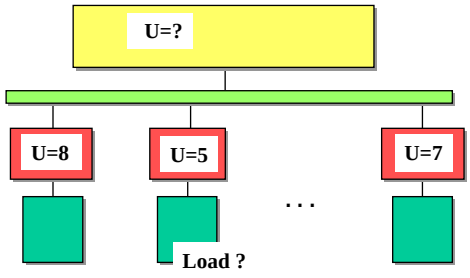


Non Coherent memory behavior

How could this happen ??

r1= 5	r1= 5	r1= 5
r2= 7	r2= 10	r2= 8
r3= 10	r3= 10	

Coherence



Coherence

- **For any execution of the program**
- **It is possible to build a hypothetical total order of access to each memory location that**
- **Would result in the actual observed result of the program**
- **Where**
  - Local partial order is the program order
  - The value returned by each read is the value written by the last write in the total order.

Coherence

- **Properties**
  - Write propagation
    - ✓ All writes become visible at some time to other processes
  - Write serialization
    - ✓ All processes see writes in the same order

Coherence

- **Need**
  - Mechanism to establish total order / global time
- **Mechanisms**
  - Bus ➔ Snoop based protocols
  - Memory ➔ Directory based protocols

Bus based coherence

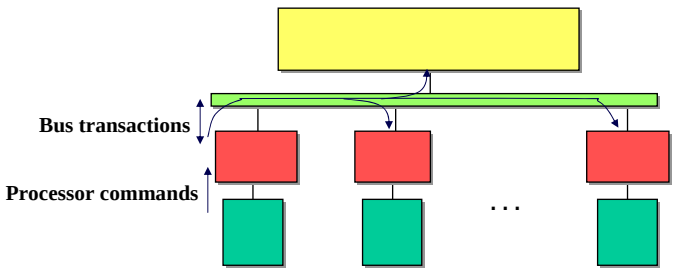
- **Bus**
  - Set of lines visible by all processors
    - ✓ Address: driven by master
    - ✓ Data: driven by master / slave / other
    - ✓ Control
      - – Driven by master (command)
      - – Driven by slave (response)
      - – Driven by other (responses)
  - Global time base
    - ✓ Bus cycles that define a global time

Snooping protocols

- **All processors listen to all bus transactions**
- **Order of writes on the bus defines the total order of writes**
  - Write propagation
  - Invalidation
- **Cache state transitions diagram**
  - Consider events from processor and from bus
  - Example: copyback caches, MSI protocol
    - ✓ Multiple readers, single writer copies of a block in cache(s)

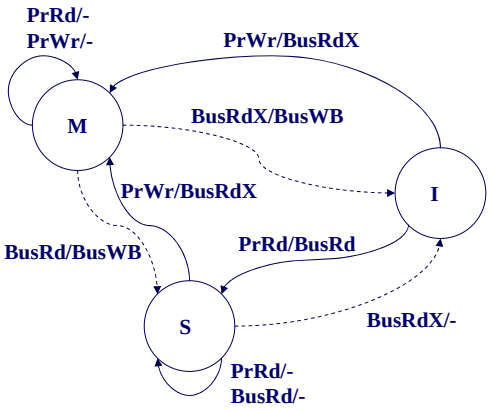
MSI

- **Processor commands**
  - PrRd
  - PrWr
- **Bus transactions**
  - BusRd: want block. Will not modify it
  - BusRdX: Want block to modify.
  - BusWB: write back block



MSI

- **Notation controller state transition**
  - command (form processor or bus) / Bus transaction
- **States**
  - Invalid
  - Shared
  - Modified



Bus transfers

- **Bus lines driven by other caches**
  - Data lines
    - ✓ cache to cache transfers
  - Control
    - ✓ Abort command / error ➔ retry
    - ✓ I am providing data (Memory don't do it)
- **Other bus transactions**
  - BusUpgr: Upgrade from shared to Modified
    - ✓ Reduce traffic

Directory based coherency

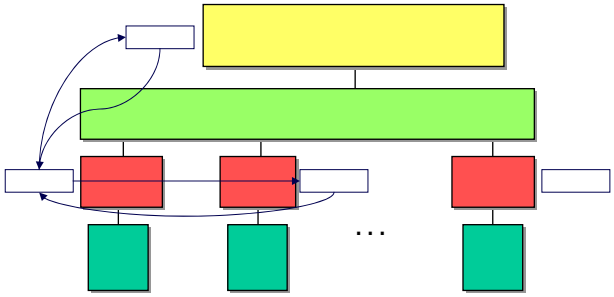
- **Interconnect:**
  - Many links. No longer a single bus
  - Point to point transactions
- **Responsibility to define global order**
  - Memory

Directory protocols

- **System state**
  - Caches: local state information
    - ✓ Same as in bus based protocols
  - Memory: Global state information ➔ directory
    - ✓ ≈ union of local states
- **Protocol**
  - Local and directory state transitions
    - ✓ Fired by processor / network transactions
    - ✓ Firing new transactions

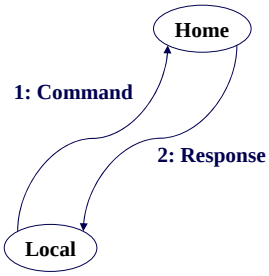
Directory protocols

- **Processor commands**
  - PrRd
  - PrWr
- **States**
  - Invalid
  - Shared
  - Modified
- **Protocol**
  - Sequence of request – response transactions involved in state transition
    - ✓ Explicit requests
    - ✓ Non atomic: risk of race conditions
  - Huge number of possibilities
    - ✓ Number of transactions in critical path?



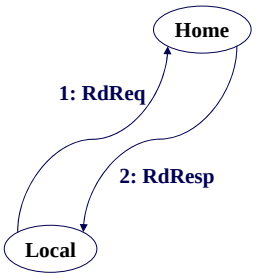
Directory protocols: transactions

- **Operation**
  - Non shared blocks (read/write)
- **Network transactions**
  - Commands
    - ✓ RdReq
    - ✓ RdXReq
    - ✓ WBReq
  - Responses
    - ✓ RdResp
    - ✓ RdXResp



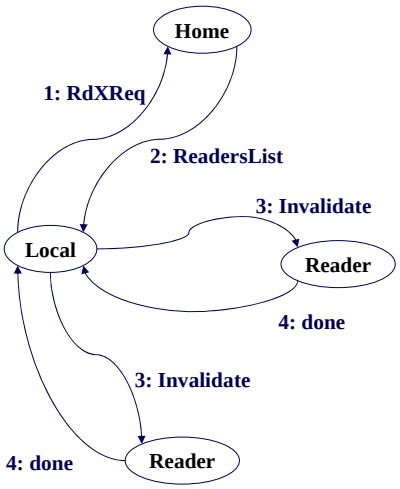
Directory protocols: transactions

- **Operation**
  - Get shared block for read
- **Network transactions**
  - Commands
    - ✓ RdReq
  - Responses
    - ✓ RdResp



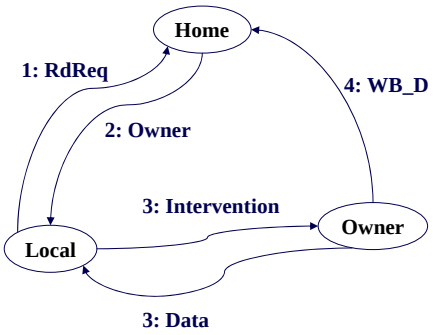
Directory protocols: transactions

- **Operation**
  - Get shared block for write
- **Network transactions**
  - Commands
    - ✓ RdXReq
    - ✓ Invalidate
  - Responses
    - ✓ ReadersList
    - ✓ Done



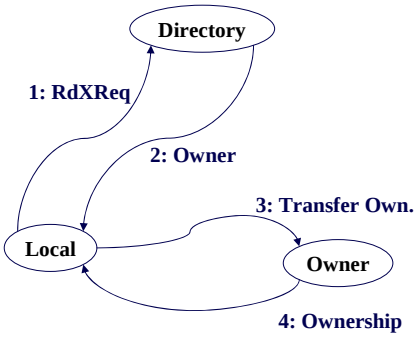
Directory protocols: transactions

- **Operation**
  - Get Modified block for read
- **Network transactions**
  - Commands
    - ✓ RdReq:
    - ✓ Intervention: Downgrade, send it to me & WB
  - Response
    - ✓ Owner
    - ✓ Data
    - ✓ WB_Downgraded



Directory protocols: transactions

- **Operation**
  - Get Modified block for write
- **Network transactions**
  - Commands
    - ✓ RdXReq
    - ✓ TransferOwnership
  - Response
    - ✓ Owner
    - ✓ Ownership



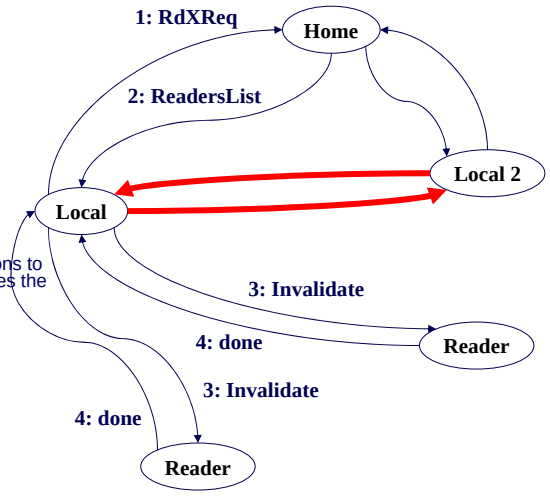
Directory protocols

- **Time**
  - Non atomic sequence of transactions
  - Establishment of global time?
    - ✓ When does owner change?
  - What if
    - ✓ Get shared block for write: sharer sends simultaneously a RdXReq
    - ✓ Get shared block for write: sharer sends simultaneously an UpgradeReq
    - ✓ …

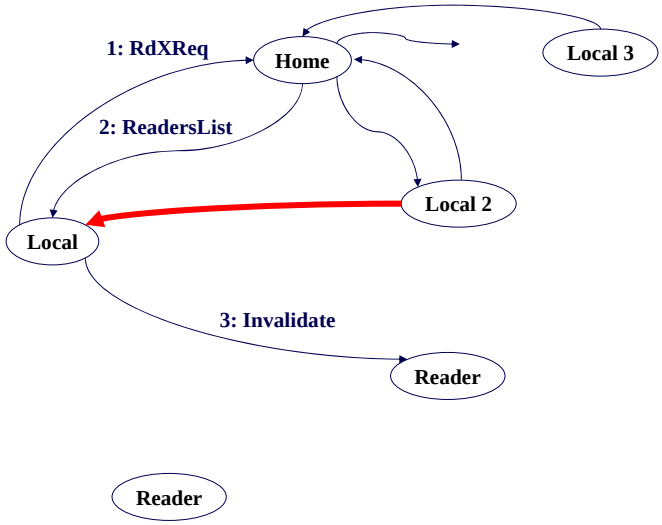
    … be safe, slow down.
  - Temporary states
    - ✓ *State transition diagram for directory and nodes in the above protocol*

Directory protocols: safe transactions

- **Slow down on**
  - Get shared block for write
  - Do not serve "later" interventions to local before it actually completes the write.

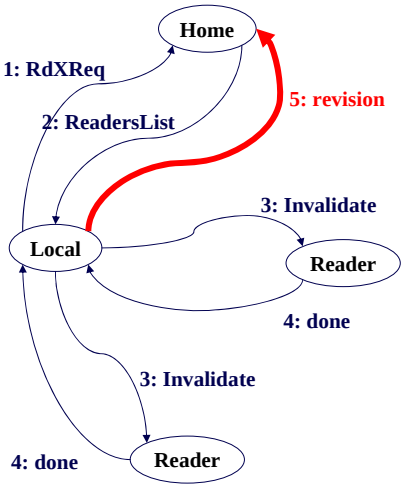


Directory protocols: safe transactions



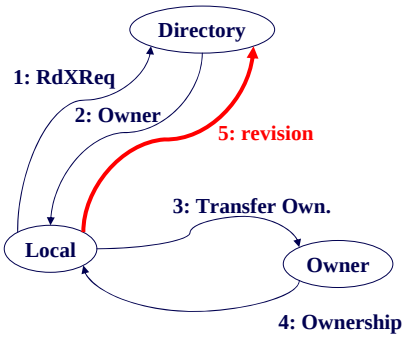
Directory protocols: safe transactions

- **Slow down on**
  - Get shared block for write
  - Extremely conservative: Avoid arrival of "later" interventions to local before it actually completes the write.



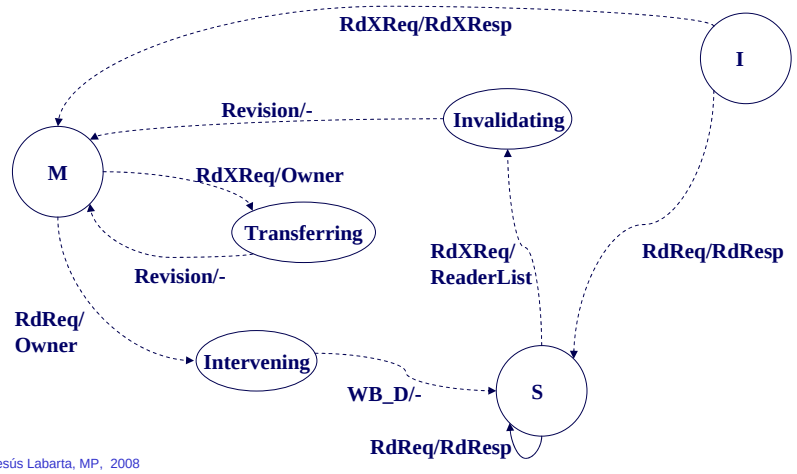
Directory protocols: safe transactions

- **Slow down on**
  - Get Modified block for write
  - Extremely conservative: Avoid arrival of "later" interventions to local before it actually completes the write.



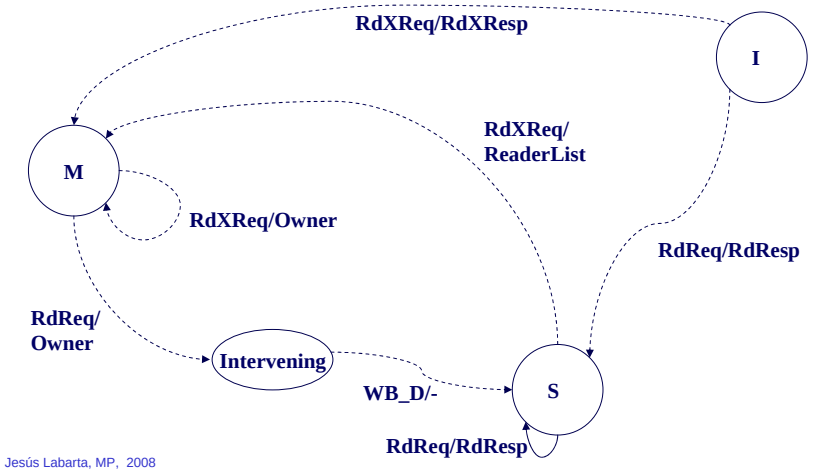
Directory protocols: states/transitions

- **Directory automata (extremely conservative)**



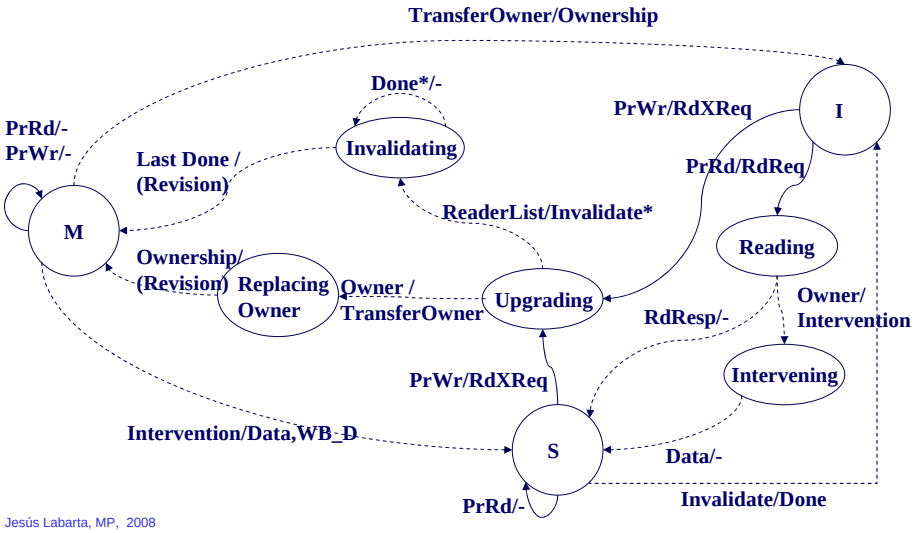
Directory protocols: states/transitions

- Directory automata



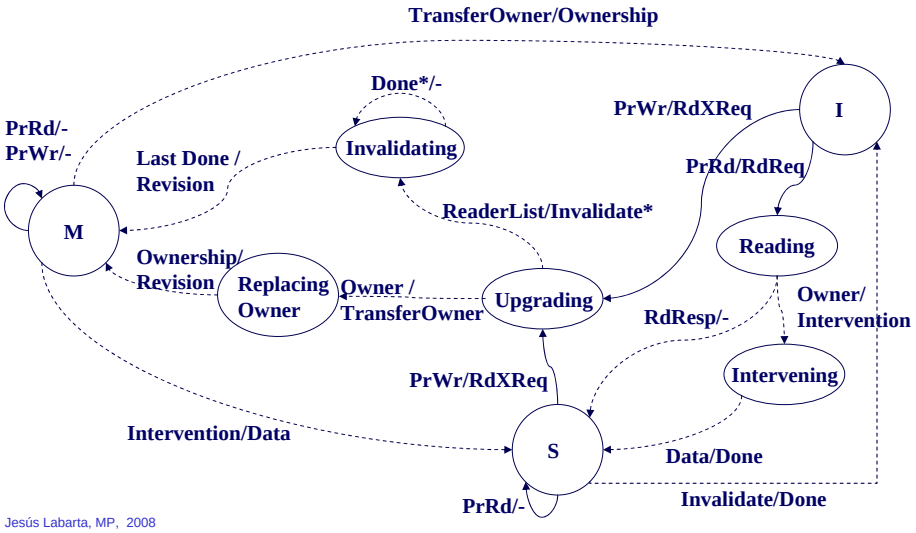
Directory protocols: states/transitions

- Local automata



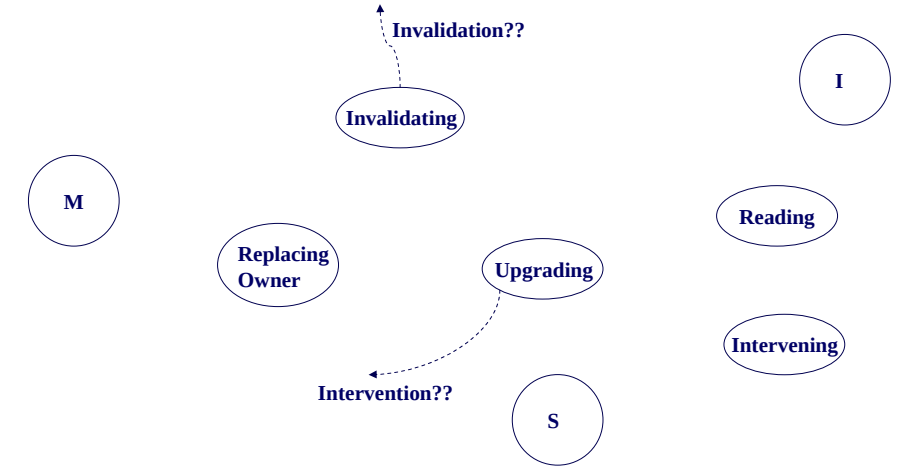
Directory protocols: states/transitions

- Local automata



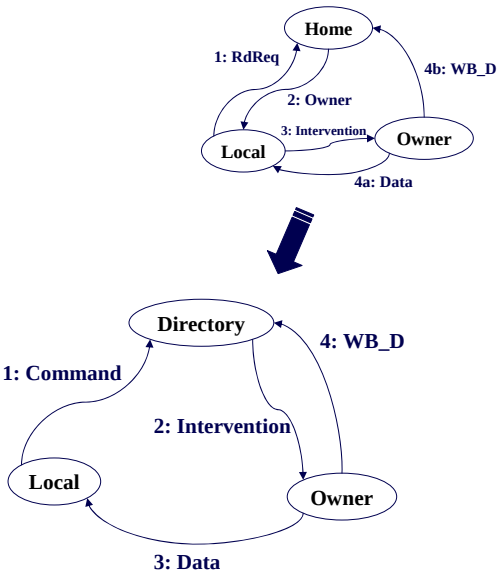
Directory protocols: states/transitions

- Local automata



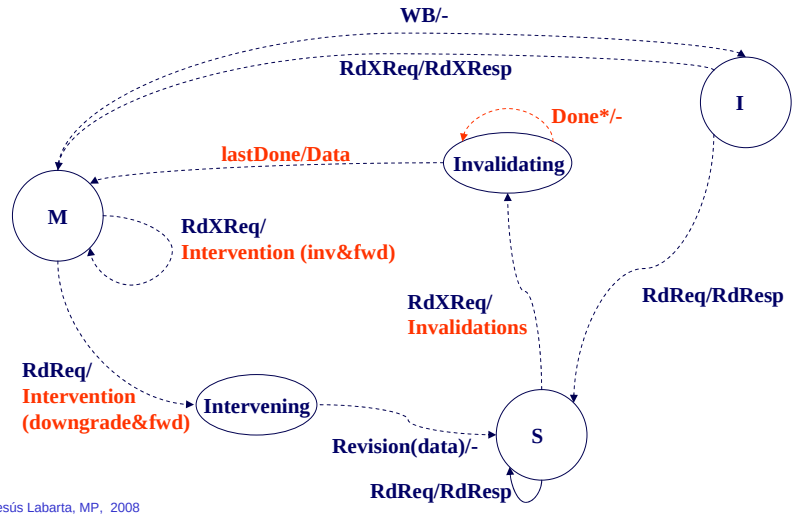
Options / optimizations

- **Three message miss protocols**
  - Reply forwarding
  - Reduce messages in critical path
  - No longer request response
  - Example
    - ✓ Get modified block for read



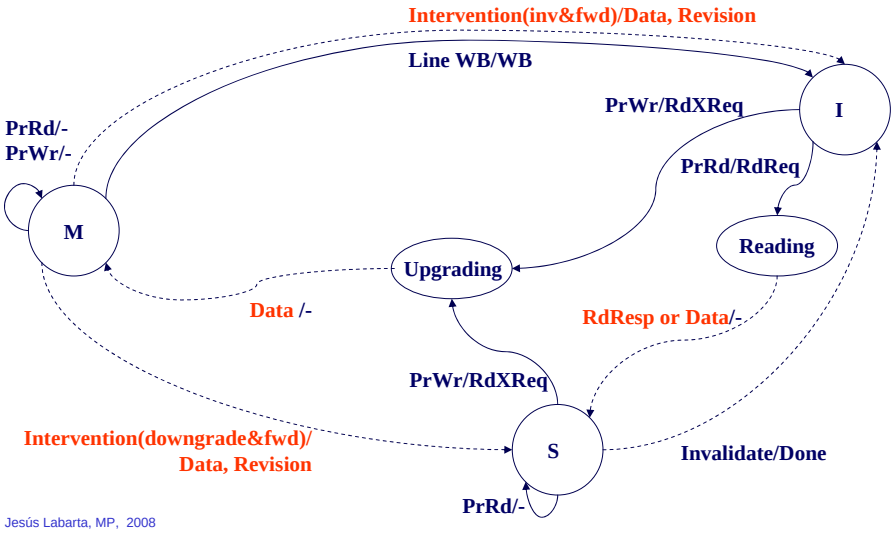
Request/Reply forwarding: states/transitions

- **Directory automata**



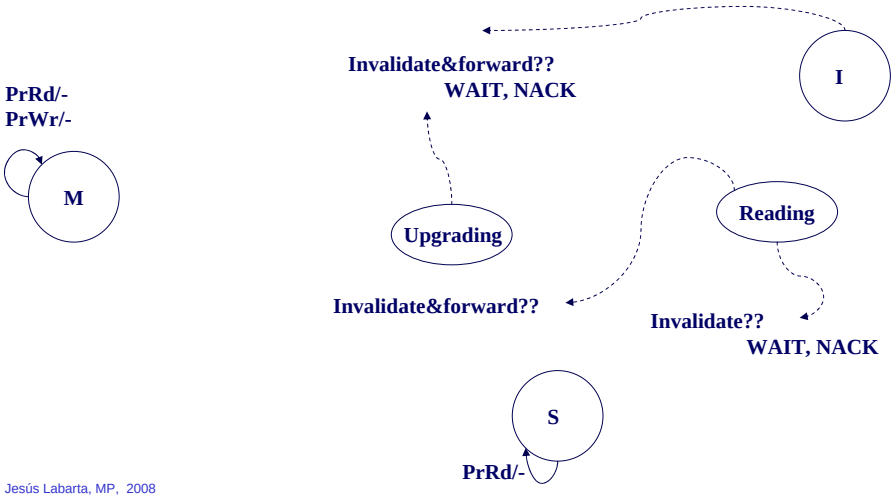
Reply forwarding: states/transitions

- Local automata



Reply forwarding: states/transitions

- Local automata



Directory

- **Centralized flat directory**
  - Full vector bit
    - ✓ 1 presence bit per processor + 1 dirty bit
  - Hierarchical
- **Distributed list**
  - Directory: head of list
  - Every node block: pointer to next sharer
  - Features
    - ✓ Sequential search
    - ✓ Limited space requirement

Options / optimizations

- **Replacement hint**
  - Notify replacement of a shared block to avoid unnecessary invalidations

Consistency

- **About:**

  Order between accesses to

  different memory locations

Consistency

- **The problem**
  - A, Flag=0

```
   P1          P2
A= 1        While (flag==0);
flag= 1     Print A;          Printed value: ?
```

  - A, B=0

```
   P1          P2          P3
A = 1       u = A       v = B       u,v,w: ?
            B = 1       w = A
```

Consistency

- **The problem**
  - A, Flag=0

```
   P1            P2
A= 1          While (flag==0);
Flag= 1       Print A;
```

Printed value: 0 ?

  - A, B=0

```
   P1          P2          P3
A = 1       u = A       v = B
            B = 1       w = A
```

u,v,w: 0, 0, 1?

u,v,w: 1, 1, 1?

u,v,w: 1, 1, 0?

Consistency

- **Memory consistency model**
  - Constraints on the order in which memory operations must appear to be performed with respect to one another.
- **Sequential consistency**
  - Effect as if:
    - ✓ Same interleaving of individual accesses seen by all processors.
    - ✓ Respects program order for each processor.

Sequential consistency

- **Sequential consistency requires**
  - Write atomicity
- **Sufficient conditions**
  - Every process issues memory operations in program order
  - After issuing a write, a process waits for it to complete before issuing the next operation
  - After issuing a read, a process waits for it to complete and for the write whose value is being returned to complete before issuing the next operation.

Sequential consistency

- **Need**
  - Mechanism to detect write completion
  - Mechanism to detect read completion
    - ✓ Data arrives
- **Sequential consistency in the snooping MSI protocol**
  - Write completion: BusRdX transaction occurs and write is performed in the cache.
  - Read completion condition: Availability of data implies write that produced it completed:
    - ✓ If read causes bus transaction, that of the write was before
    - ✓ If data was M: this processor had already completed the write
    - ✓ If data was S: this processor had already read the data (go to 1 or 2)

Sequential consistency

- **Sequential consistency in directory protocols**
  - Risks:
    - ✓ Different paths through the network. Different delays, congestions
    - ✓ Network not preserving order of messages

Sequential consistency

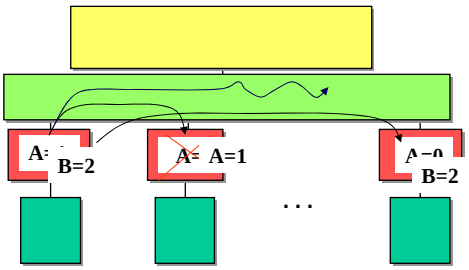
- Write completion
  - ✓ A=B=0

```
   P1
A= 1;
…;
B= 2;
```

```
   P2, P3, P4, …
while (A==0 && B==0);
print A, “,”, B;
```

Printed values ?

P2	P3	P4
1,0	1,2	0,2

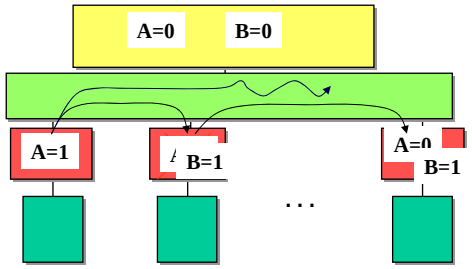


Sequential consistency

- Write atomicity
  - ✓ A=B=0

```
   P1          P2                  P3
A= 1        While (A==0);       …=A;
            B=1;                …
                                While (B==0);
                                Print A;
```

Printed value: 0 ?



Sequential consistency

- **Sequential consistency in directory protocols**
  - Write completion detection
    - ✓ Acknowledgement of invalidations
  - Write atomicity
    - ✓ Owner not allowing accesses to the new value until all acknowledges have returned.
  - Risks:
    - ✓ Concurrent transactions under:
      - – Different paths through the network. Different delays, congestions
      - – Network not preserving order of messages

Synchronization

- **Need to ensure certain order between computations**
  - Mutual exclusion
  - Point to point event
  - Collective synchronization
    - ✓ Barrier
- **Components**
  - Acquire method
    - ✓ Acquire the right to the synchronization
  - Waiting algorithm
  - Release method
    - ✓ Allow others to proceed

Synchronization

- **Waiting algorithm**
  - Busy wait
    - ✓ Faster
    - ✓ Resource consumption during wait
    - ✓ Deadlock ?
  - Blocking
    - ✓ Overhead
    - ✓ Dependence on scheduler
  - Two phase
    - ✓ Wait for a while, then block
    - ✓ Sensitive to waiting time
      - – x1 / x2 context switch overhead
    - ✓ Can frequently be controlled through an environment variable

Mutual exclusion

- **Problem with following code ?**

```
lock: ld r1, location
      st location, #1
      bnz r1, lock

Unlock: st location, #0
```


Mutual exclusion → t&s

- **Mutual exclusion based on atomic modification of variable**

```
lock: t&s r1, location
      bnz r1, lock

Unlock: st location, #0
```

- **Atomicity**
  - Single processor: 1 instruction
  - Multiprocessor: atomic memory read and update
    - ✓ Lock bus → inefficient
    - ✓ Read exclusive and do not release till updated

Mutual exclusion: performance

- **Goals**
  - Low latency
  - High throughput
  - Scalability
  - Low storage cost
  - Fairness

t&s performance

- **Latency**
  - Low
  - Cached locks → faster reuse
- **Throughput**
  - Write every iteration by all threads → a lot of traffic and invalidations
- **Scalability: poor**
- **Storage: low**
- **Fairness:**
  - No
  - Cached locks → higher probability of success

Mx algorithms

- **t&s with backoff**
  - Insert delays after unsuccessful attempts to acquire the lock
    - ✓ Not too long: would lead to idle lock while requests pending
    - ✓ Not too short: contention
  - Exponential backoff: $k \cdot c^i$
- **t&t&s**
  - Wait by using a normal load instruction
    - ✓ Lock will be cached and wait will not use system bandwidth

Mx algorithms

- **LL-SC**
  - SC will fail if other processor modifies the variable . If it succeeds the update was atomic.

```
lock: ll    r1, location
      bnz   r1, lock          Unlock: st location, #0
       r2 = f (r1)
      sc    location, r2
      beqz  lock
```

  - *f: #1* ≡ test&set
  - *f: +1* ≡ fetch&increment
  - *f: +ri* ≡ fetch&add
  - …

Mx algorithms

- **LL-SC**
  - A failing SC does not generate invalidations
    - ✓ Hardware lock flag and lock address register
    - ✓ Replacements might lead to livelock → avoid them
      - – Disallow memory references in instruction sequence *f*
      - – Do not allow memory instruction reordering across LL-SC
  - The unlock still generates an invalidation to all contending processors and their simultaneous attempt to reacquire the lock

Mx algorithms

- **Ticket lock**

```
lock: f&i   r1, counter
      while (r1!=now_serving);
```

```
Unlock: now_serving++;
```

  - Contention for now-serving after release
    - ✓ Proportional backoff ∝ (r1 – now_serving)
  - Fairness: yes

Mx algorithms

- **Array-based lock**

```
lock: f&i   r1, counter
      while (array[r1]==WAIT);
```

```
Unlock: array[r1+1]=CONTINUE;
```

  - Storage cost:
    - ✓ Array layout to avoid false sharing
    - ✓ Proportional to P
  - Fairness: yes
  - Other issues
    - ✓ Reuse of positions → Initialization ?

Point to point event synchornization

- **Normal load/sore instructions on ordinary variables**

```
signal: st  flag, #1
```

```
wait: ld r1, flag
      bz r1, wait
```

  - The variable itself can be used as flag if there is a possible value that the produced value will never have (minint, 0,…)

Barrier algorithms

- **Centralized**

```
Barrier(barr) {
   lock (barr.lock);
   if (barr.counter == 0)
      barr.flag = 0;            // reset flag if first
   mycount =  barr.counter++;
   unlock (barr.lock);
   if (mycount == P) {          // last to arrive?
      barr.counter =0;             // reset for next barrier
      barr.flag = 1;               // release waiting processors
   } else
      while (barr.flag == 0);   // busy wait for release
}
```

  - Potential of deadlock
    - ✓ Back to back barriers

Barrier algorithms

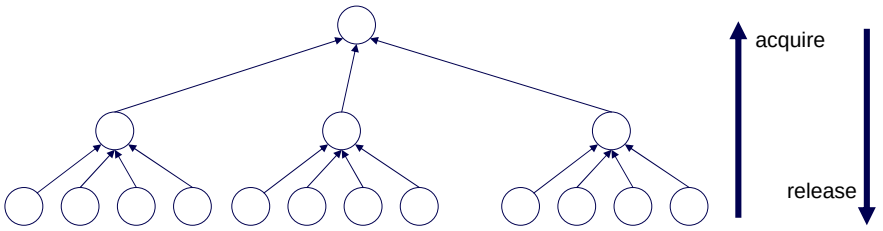
- **Centralized with sense reversal**

```
Barrier(barr) {
   local_sense = !(local_sense);
   lock (barr.lock);                          } // acquire
   mycount =  barr.counter++;
   if (barr.counter == P) {
      unlock (barr.lock);
      barr.counter =0;                        }  // release
      barr.flag = local_sense;
   } else
      unlock (barr.lock);
      while (barr.flag != local_sense);       } // wait
}
```

  - Contention on counter

Barrier algorithms

- **Tree based barriers**



  - Trade off
    - ✓ Overlap between potentially contending phases
    - ✓ Increased latency
  - Can be implemented with ordinary load/stores

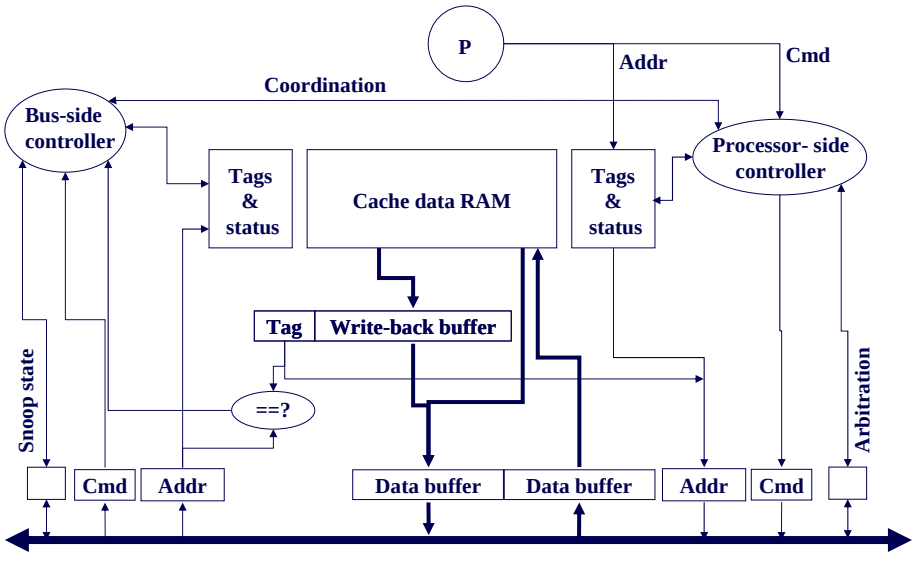
Further details

- **MESI**
- **Snooping designs**
  - Atomic transaction, non atomic global process
  - Split transaction bus

MESI

- **Performance of not sharing applications under MSI?**
  - Two transactions per modified line
    - ✓ BusRd
    - ✓ BusRdX / Upgrade
- **Exclusive state**
  - Only read copy in the system
  - Transition from Exclusive to Modified does not require bus transaction
  - BusRd fires transition to either Shared or Exclusive depending on some other cache has the block or not

Snooping cache design



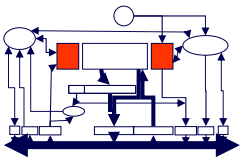
Snooping cache design

- **Replicated tags & status**
  - Concurrent access by processor and snoop controllers
  - Should be identical (~)
- **Snoop bus lines**
  - Wired OR control lines. Ex:
    - ✓ Snoop hit: the address matches a tag in some processor
    - ✓ Modified state: some processor has the line block in modified state
      - – Memory should detect it and not respond to request
    - ✓ Valid snoop result: all processors performed their snoop check and the other two lines are valid

Snooping cache design

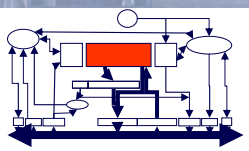
- **Write backs**
  - Proc. Controller transfers dirty line to write-back buffer
  - Requests BusRd(X) transaction
  - Requests WB transaction
  - Snoop controller: if transaction affects line
    - ✓ respond appropriately and provide data
    - ✓ Proc controller: Cancel WB request

Snooping cache design



- **Coordination**
  - Update of snoop and processor tags
    - ✓ I➔S,E
      - – No conflict. Processor is waiting
    - ✓ Invalidations
      - – Simultaneous
        - » Blocking processor
        - » Delay transaction till achieved
      - – Delayed
        - » Queue request to processor controller
        - » Coherence and consistency ?

Snooping cache design



- **Coordination**
  - Access to Cache data
    - ✓ On M➔S transition Snoop controller needs access to data
    - ✓ Delay transaction till access achieved
  - Non atomic state transitions
    - ✓ While posting a request, a transaction may be seen that affects the block and requires a change in state and type of request. Ex:
      - – WB posted and BusRdX observed ➔ provide data, Cancel WB request
      - – Upgrade posted and other Upgrade observed ➔ Invalidate, post BUsRdX
    - ✓ Temporary states may be encoded in communication variables between both controllers

Snooping cache design

- **Coherence and consistency**
  - When can a write be performed in the local cache?
    - ✓ Bus granted for transaction
      - – The actual block invalidation in other processor may take place later
      - – Commitment ↔ completion

Snooping cache design

- **Resources**
  - Buffers
    - ✓ Write back buffers
    - ✓ Notifications from snoop to processor controller
    - ✓ …
  - Require associative comparisons
  - If exhausted may cause deadlock

Snooping cache design

- **Deadlock**
  - No system activity
  - It is necessary to continue servicing incoming transactions while posting requests
  - Resource limitations may cause deadlocks
    - ✓ Avoidance: NACKs

Snooping cache design

- **Livelock**
  - System activity but no real progress
  - Ex: complete write immediately after entering Modified state

Snooping cache design

- **Starvation**
  - Some processors do not progress
  - Fair scheduling policies
    - ✓ FIFO arbiter
    - ✓ Threshold on count of denied accesses
  - May appear at higher levels (synchronization algorithms,…)

Snooping cache design

- **Bus synchronization**
  - Synchronous
    - ✓ Master clock
    - ✓ Predefined timing between requests and responses
    - ✓ Conservative timings: designed for worst case
  - Asynchronous
    - ✓ Handshake protocol

Snooping @ Multilevel caches

- **State @ L1**
  - Determines the action of the processor on hits
- **State @ L2**
  - Determines the reaction of the snoop controller to the bus transactions
- **The snoop controller must respond for the node**
  - Need to coordinate state changes between L1 and L2
  - Need to maintain inclusion

Snooping @ Multilevel caches

- **Inclusion property**
  - If block in L1 then also in L2
  - If block is in owned state in L1 then also in L2
- **Maintaining inclusion**
  - Not immediate
    - ✓ L1 and L2 see different access patterns ➔ may take different replacement decisions
      - – L1 set associative with LRU can violate inclusion
      - – Multiple independent L1 caches
      - – Different cache block sizes

Snooping @ Multilevel caches

- **Maintaining inclusion**
  - Guaranteed if
    - ✓ direct L1
    - ✓ Incoming blocks are put in both L1 and L2
    - ✓ same L1 and L2 block size
    - ✓ #lines in L1 ≤ #sets in L2

Snooping @ Multilevel caches

- **Enforcing inclusion by using coherence mechanisms**
  - Replacement in L2
    - ✓ invalidation/flush sent to L1
  - Snoop actions @ L2
    - ✓ Need to invalidate L1 as reaction to bus transaction
      - – Send intervention for every transaction relevant to L2
      - – Keep inclusion bit for each line
  - Processor writes
    - ✓ Write through
    - ✓ L2 state: modified but stale
      - – Request block to L1 if needed

Snooping @ Split transaction bus

- **Split transaction bus**
  - Transactions split into two independent phases
    - ✓ Request and response
    - ✓ Several ongoing requests concurrently

Snooping @ Split transaction bus

- **Split transaction bus issues**
  - Split snoop
    - ✓ Within request or response?
  - Flow control
    - ✓ limited buffers for incoming requests and responses
  - Out of order responses?
  - Conflicting requests
    - ✓ at least one of them a write
    - ✓ Not possible to modify the later requests. Need to properly handle coherence states, transitions and responses
  - Trade-off
    - ✓ Performance ↔ resources (buffers) ↔ complexity ↔cost

Example split transaction snooping

- **Options**
  - Maximum outstanding requests: 8
  - Conflicting requests: Not allowed
  - Flow control: NACK
  - Out of order responses: yes
  - Split snoop
    - ✓ Total order established by request phase
    - ✓ Snoop results presented
      - – Modified in request phase
      - – Shared in response phase ← Read merge
  - 128 bytes blocks

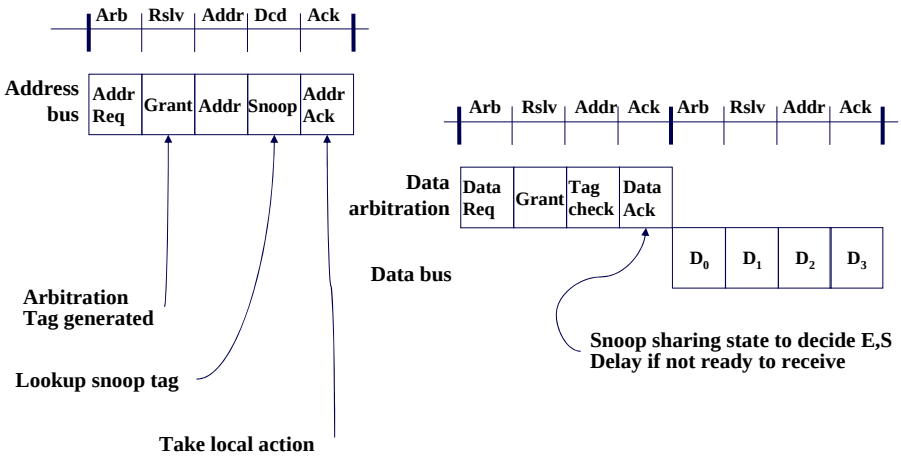
Example split transaction snooping

- **Split transaction phases**
  - Request
  - Response
    - ✓ Arbitration
    - ✓ Actual data response

Example split transaction snooping

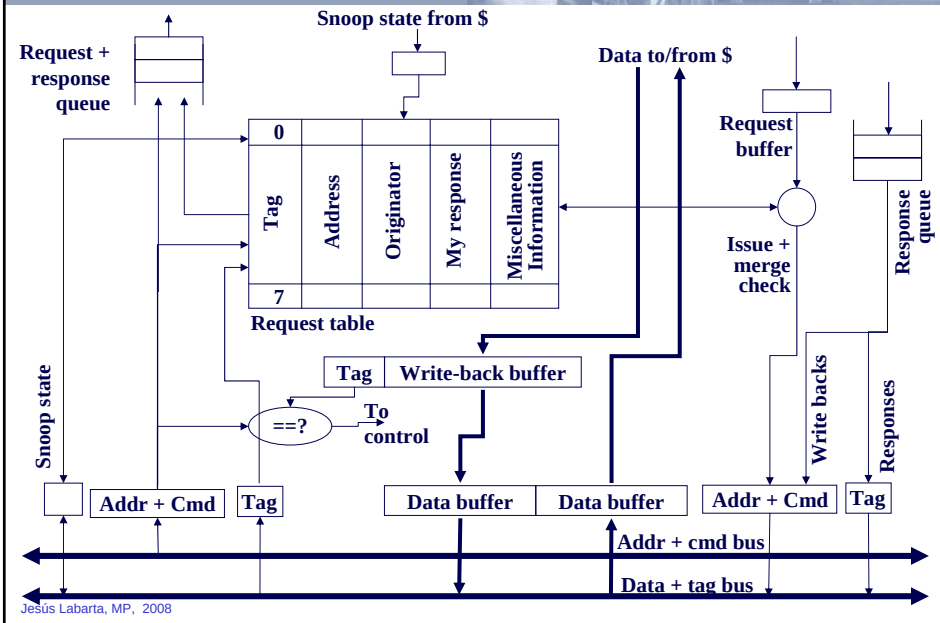
- **Bus**
  - Request bus:
    - ✓ command & address
    - ✓ Request identification
      - – 3 bits tag
    - ✓ 5 cycles: arbitration, resolution, address, decode, acknowledge
  - Response bus:
    - ✓ Data (256bits) & tag
    - ✓ 4 cycles

Snooping @ Split transaction bus



Take local action
Drive snoop lines: tell memory whether it has to respond or not
Delay if not time to check/NACK
Note if response needed
Withdraw request if conflicting with previous

Snooping @ Split transaction bus



Snooping @ Split transaction bus

- **Request issue**
  - Wait if conflicting with one in Request table
- **Request merge**
  - Read request for same block as one BusRd entry in Request table
    - ✓ Set bit in Request table: I want the data
    - ✓ Bit in Request table: I generated the request
      - – If not, I activate the snoop sharing line during the data response phase so that block is Shared instead of Exclusive
- **Write completion**
  - Write commitment in the last cycle of the request phase

Snooping @ Split transaction bus

- **Snoop result only in response phase ?**
  - Would allow faster request phase
  - Memory
    - ✓ Start servicing every request
    - ✓ Abort service if observed response from a cache
    - ✓ When responding, observe snoop lines (wait if inhibit) if dirty line activated cancel response

Snooping @ Split transaction bus

- **Sequential consistency ?**
  - Global order established by request phase
  - Write commit substitute for completion
  - Completion may take place much later than commitment
  - At each processor, serve incoming queue before completing a read miss or write that generates a bus transaction

Snooping @ Split transaction bus

- **In order responses ?**
  - Potential performance loss
    - ✓ Access patterns to interleaved memory modules
      - – 2 requests to module A followed by a request to module B
  - Potential to support conflicting requests

Snooping @ Split transaction bus

- **Write Back?**
- **Shared lines**
  - Two logical busses may share physical lines
    - ✓ Lines not used in one cycle by one bus may be used by the other
  - Pipeline stages skewed in order not to conflict
    - ✓ Must stay synchronized. Inhibits affect both busses
  - Complexity vs. cost