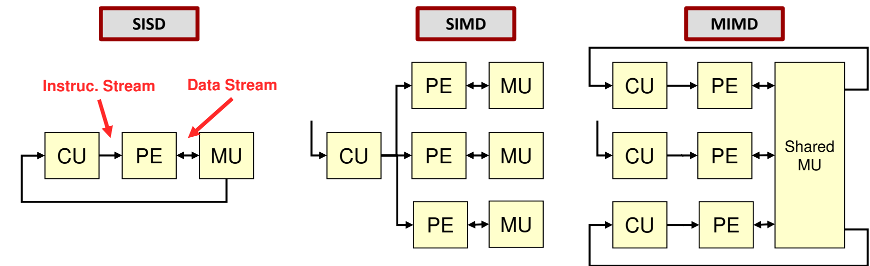
EE 457 Unit 10

Parallel Processing

Cache Coherency

Parallel Processing Paradigms

- SISD = Single Instruction, Single Data
  - Uniprocessor
- SIMD = Single Instruction, Multiple Data
  - Multimedia/Vector Instruction Extensions, Graphics Processor Units (GPU's)
- MIMD = Multiple Instruction, Multiple Data
  - CMP, CMT, Parallel Programming



SIMD Execution

- Given 4 processing elements we can use the same code to perform only ____________ iterations
  - Addressing is managed separately for each processing element so that it receives different data elements to operate on

```
for(i=0; i < 10,000; i++)
  A[i] = B[i] + C[i];
```

Sequential Execution
(10,000 iterations)

```
for(i=0; i < _______; i++)
  for(j=0; j < ___; j++)
    A[4*i+j] = B[4*i+j] + C[4*i+j];
```

Equivalent Execution – Still 10,000 iterations
(j Processing Elements)

```
#pragma vectorize v=[0..3]
for(i=0; i < ________; i=i++)
  A[4*i+v] = B[4*i+v] + C[4*i+v];
```

Vectorized Execution
(Each PE operates in parallel
requiring only _______ iterations)

SIMT Execution

- Each thread uses its unique ID to execute the same code but on different data
  - Each thread has its own register set / addressing scheme
- Partial sums can be generated independently
- When all threads are done (synchronization!) we can combine results
  - Requires ____________ between units

```
for(i=0; i < 10,000; i++)
  sum = sum + A[i];
```

Sequential Execution
(10000 iterations)

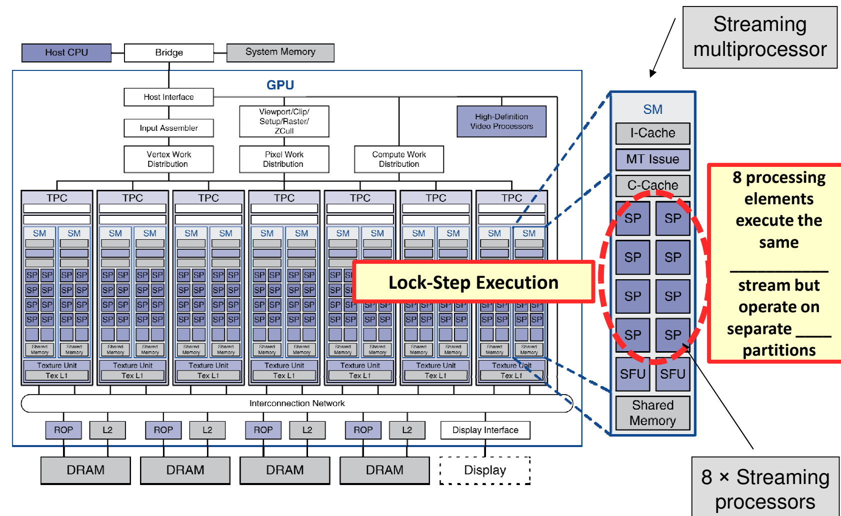
```
for(t=0; t < 10; t++)
  for(i=0; i < 1,000; i++)
    sum = sum + A[1000*t + i];
```

Equivalent Execution
(10 * 1000 iterations)

```
#pragma parallel t=[0..9]
for(i=0; i < 1,000; i++)
  sum[t] = sum[t] + A[1000*t + i];

// combine each threads results
// requires communication between threads
for(t=0; t < 10; t++)
  sum += sum[t];
```

Parallel Execution in 10 Threads
each with its own value of t
(1000 iterations per thread)

SIMT Example: NVIDIA Tesla GPU

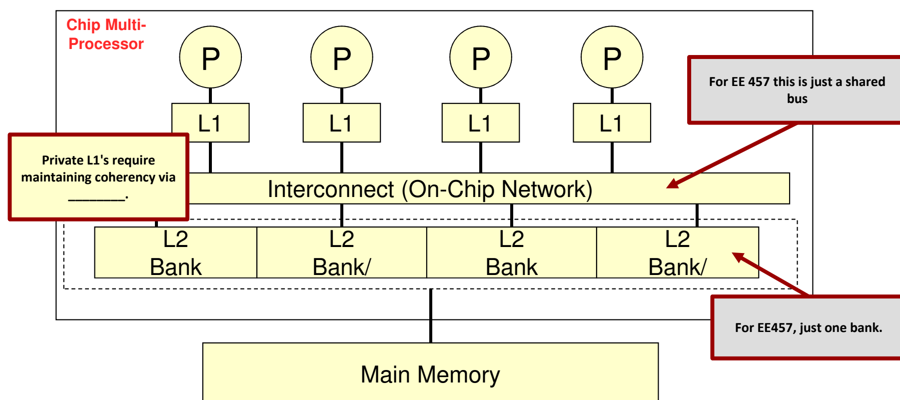


H&P, CO&D 4th Ed. Chapter 7 — Multicores, Multiprocessors, and Clusters — 5

MIMD

- An MIMD machine consisting of several SISDs yields higher performance when different tasks require execution
- How do parallel processors…
  - Share data?
  - Coordinate and synchronize?
    - In MIMD, we no longer run in lock-step but execute different tasks at their own rate requiring coordination through synchronization
- Two communication paradigms
  - _______________ (can each access the same address space)
  - _______________ (private address spaces per process/thread with explicit messages passed between them)

Typical CMP Organization



Definitions

- Multiprogramming
  - Running multiple independent programs using time-sharing on the same processor
- Multiprocessing
  - Running multiple independent programs on a multiprocessor
- Multitasking
  - Splitting a single application into multiple tasks which can be run on a time-shared uniprocessor or on a multiprocessor
- Multithreading
  - Same as multitasking; however tasks are executed by "lightweight" processes or "threads" within a **single** process

ISCA '90 Tutorial "Memory System Architectures for Tightly-coupled Multiprocessors", Michel Dubois and Faye A. Briggs © 1990.

Programming Model

- Applications are partitioned into a set of cooperating processes
- Processes can be seen as “virtual processors”
  - Usually there are many more processes than processors and time-sharing is required
- Processes may communicate by passing messages
  - Usually done by shared mailboxes (shared memory variables) or shared regions of memory in a shared memory system
  - Interprocessor interrupts or network I/O in a message passing system
- For shared memory systems, synchronization protocols must be careful followed to avoid read-modify-write race conditions
- Scheduling: Binding processes to processors

ISCA ‘90 Tutorial “Memory System Architectures for Tightly-coupled Multiprocessors”, Michel Dubois and Faye A. Briggs © 1990.

Difficulties in Exploiting MIMD

- ________________
  - Synchronization, locks, race conditions, etc
- In many cases, parallel programming requires a fair amount of knowledge of the underlying ___________ to achieve ______________
- Limitation of speedup due to __________ (i.e. the portion of code that is NOT parallelized)
  - Sequential job take 100 Time Units
  - 80 Time units are parallelized to 10 processors
  - New Exec. Time = _______________
  - Speedup = ______________
    - Compared to linear speedup expectation of 10 proc. => 10x speedup)

Synchronization

- Example: Suppose we need to sum 10,000 numbers on 10 processors. Each processor sums 1,000 at its own pace and then need to combine results
- We need to wait until the 10 threads have completed to combine results
- This is an example of a _______ synchronization where all threads must check in and reach the “________” sync point *before* any thread may continue
  - No one shall execute beyond the barrier until all others reach that point
- To implement this we keep a count and increment it atomically

________ must be performed atomically.

```
_______(N)
{
 count = count+1;
 if(count == N)
   - resume all
     processes
   - count = 0
 else
   - block task and
     place in
     barrier queue
}
```


Problem of Atomicity

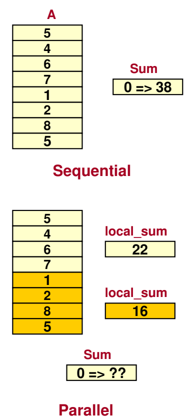
- Sum an array, A, of numbers {5,4,6,7,1,2,8,5}
- Sequential method

  for(i=0; i < 7; i++) { sum = sum + A[i]; }
- Parallel method (2 threads with ID=0 or 1)

  for(i=ID*4; i < (ID+1)*4; i++) {

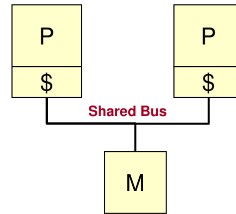
  local_sum = local_sum + A[i]; }

  sum = sum + local_sum;
- Problem
  - Updating a shared variable (e.g. sum)
  - Both threads read sum=0, perform sum=sum+local_sum, and write their respective values back to sum
  - Sum ends up with only a partial sum
  - Any read/modify/write of a shared variable is susceptible
- Solution
  - Atomic updates accomplished via some form of locking



Atomic Operations

- Read/modify/write sequences are usually done with separate instructions
- Possible Sequence:
  - P1 Reads sum (lw)
  - P1 Modifies sum (add)
  - P2 Reads sum (lw)
  - P1 Writes sum (sw)
  - P2 uses old value…
- Partial Solution: Have a separate flag/"lock" variable (0=Lock is free/unlocked, 1 = Locked)
- Lock variable is susceptible to same problem as sum (read/modify/write)
- Hardware has to support some kind of instruction to implement atomic operations usually by not releasing bus between read and write



Thread 1:	Thread 2:
Lock L	Lock L
Update sum	Update sum
Unlock L	Unlock L

Locking/Atomic Instructions

- TSL (Test and Set Lock)
  - tsl reg, addr_of_lock_var
  - Atomically stores const. '1' in lock_var value & returns lock_var in reg
    - Atomicity is ensured by HW not releasing the bus during the RMW cycle
- LL and SC (MIPS & others)
  - Lock-free atomic RMW
  - LL = Load Linked
    - Normal lw operation but tells HW to track any external accesses to addr.
  - SC = Store Conditional
    - Like sw but only stores if no other writes since LL & returns 0 in reg. if failed, 1 if successful

```
LOCK:    TSL   $4,lock_addr
         BNE   $4,$zero,LOCK
         return;

UNLOCK:  sw    $zero,lock_addr
```

```
         LA    $8,lock_addr
LOCK:    ADDI  $9,$0,1
         LL    $4,0($8)
         SC    $9,0($8)
         BEQ   $9,$zero,LOCK
         BNE   $4,zero,LOCK
```

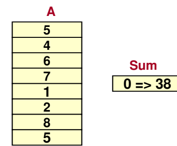
```
         LA    $t1,sum
UPDATE:  LL    $5,0($t1)
         ADD   $5,$5,local_sum
         SC    $5,0($t1)
         BEQ   $5,$zero,UPDATE
```

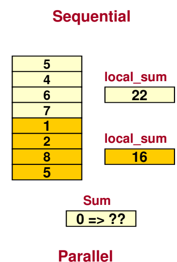

Solving Problem of Atomicity

- Sum an array, A, of numbers {5,4,6,7,1,2,8,5}
- Sequential method

  for(i=0; i < 7; i++) { sum = sum + A[i]; }
- Parallel method (2 threads with ID=0 or 1)

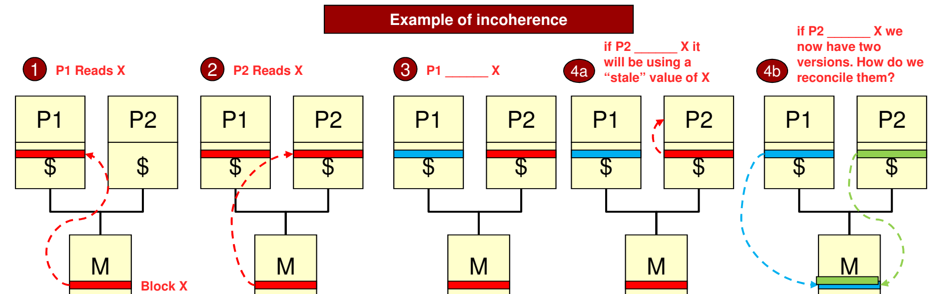
```
lock L;
for(i=ID*4; i < (ID+1)*4; i++) {
  local_sum = local_sum + A[i]; }

getlock(L);
sum = sum + local_sum;
unlock(L);
```





Cache Coherency

- Most multi-core processors are shared memory systems where each processor has its own cache
- Problem: Multiple cached copies of same memory block
  - Each processor can get their own copy, change it, and perform calculations on their own different values…INCOHERENT!
- Solution: __________ caches…

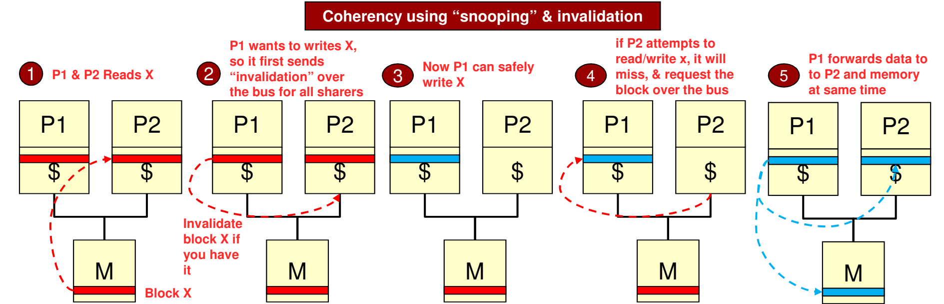


Snoopy or Snoopy

Solving Cache Coherency

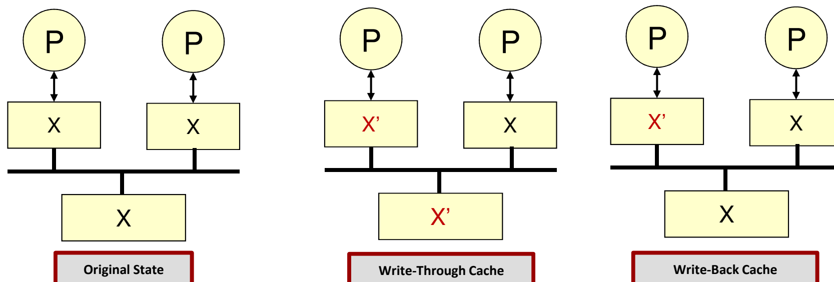
- If no writes, multiple copies are fine
- Two options: When a block is modified
  - Go out and update everyone else's copy
  - Invalidate all other sharers and make them come back to you to get a fresh copy
- "Snooping" caches using invalidation policy is most common
  - Caches monitor activity on the bus looking for invalidation messages
  - If another cache needs a block you have the latest version of, forward it to mem & others

Coherency using "snooping" & invalidation



Coherence Definition

- A memory system is coherent if the value returned on a Load instruction is always the value given by the latest Store instruction with the same address
- This simple definition allows to understand the basic problems of private caches in MP systems



ISCA '90 Tutorial "Memory System Architectures for Tightly-coupled Multiprocessors", Michel Dubois and Faye A. Briggs © 1990.

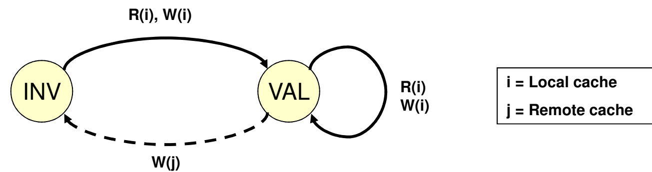
Write Through Caches

- The bus interface unit of each processor "watches" the bus address lines and invalidates the cache when the cache contains a copy of the block with modified word
- The state of a memory block b in cache i can be described by the following state diagram
  - State INV: there is no copy of block b in cache i or if there is, it is invalidated
  - State VAL: there is a valid copy of block b in cache i

ISCA '90 Tutorial "Memory System Architectures for Tightly-coupled Multiprocessors", Michel Dubois and Faye A. Briggs © 1990.

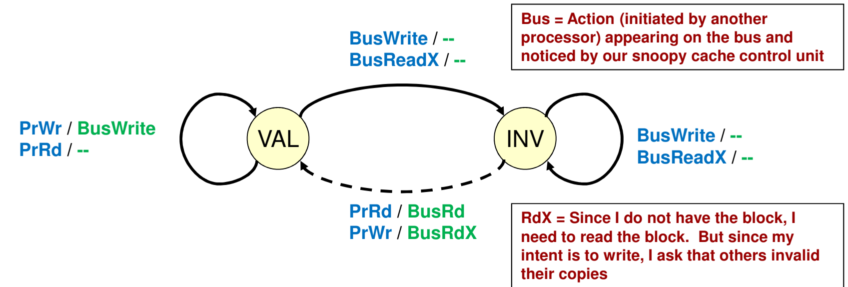
Write Through Snoopy Protocol

- R(k): Read of block b by processor k
- W(k): Write into block b by processor k
- Solid lines: action taken by the local processor
- Dotted lines: action taken by a remote processor (incoming bus request)



Bus vs. Processor Actions

- Cache block state (state and transitions maintained for each cache block)
  - Format of transitions: Input Action / Output Action
  - Pr = Processor Initiated Action
  - Bus = Consequent action on the bus



Michel Dubois, Murali Annavaram and Per Stenström © 2011.

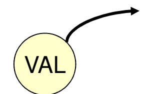
Action Definitions

Acronyms	Description
PrRd	Processor Read
PrWr	Processor Write
BusRd	Read request for a block
BusWrite	Write a word to memory and invalidate other copies
BusUpgr	________________________________
BusUpdate	Update other copies
BusRdX	________________________________
Flush	Supply a block to a requesting cache
S	Shared line is activated
~S	Shared line is deactivated

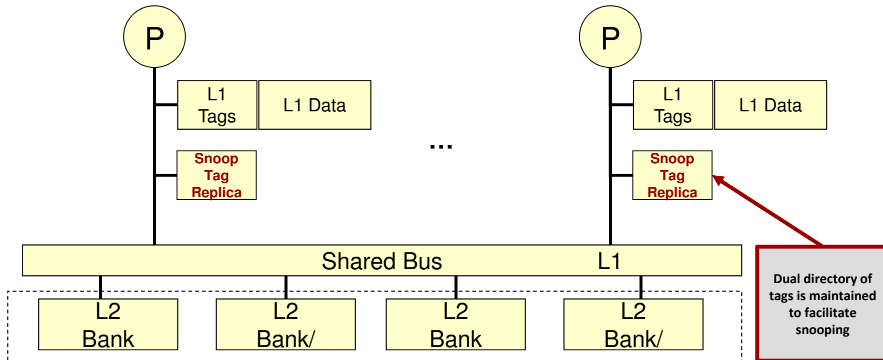
Michel Dubois, Murali Annavaram and Per Stenström © 2011.

Cache Block State Notes

- Note that these state diagrams are high-level
  - A state transition may take multiple clock cycles
  - The state transition conditions may violate all-inclusive or mutually-exclusive requirements
  - There may be several other intermediate states
  - Events such as replacements may not have been covered



Coherence Implementation

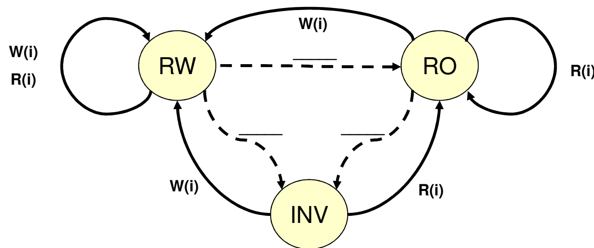


Write Back Caches

- Write invalidate protocols ("Ownership Protocols")
- Basic 3-state (MSI) Protocol
  - I = INVALID: Replaced (not in cache) or invalidated
  - RO (Read-Only) = __________: Processors can read their copy. Multiple copies can exist. Each processing having a copy is called a "Keeper"
  - RW (Read-Write) = __________: Processors can read/write its copy. Only one copy exists. Processor is the "Owner"

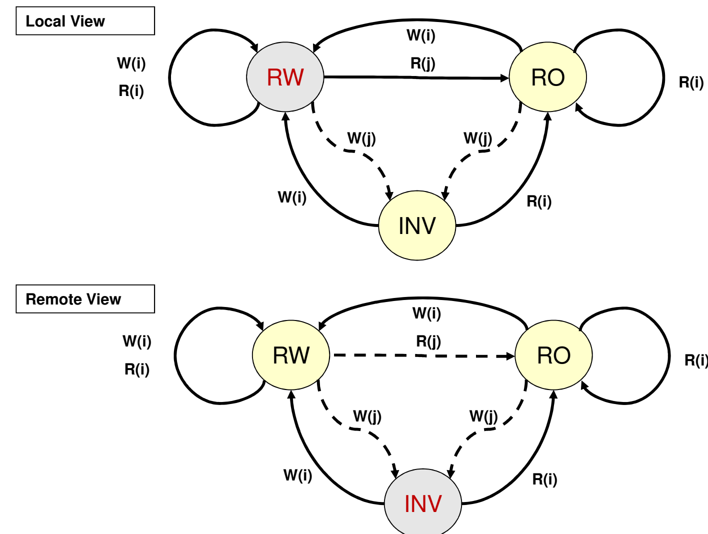
ISCA '90 Tutorial "Memory System Architectures for Tightly-coupled Multiprocessors", Michel Dubois and Faye A. Briggs © 1990.

Write Invalidate Snoopy Protocol



ISCA '90 Tutorial "Memory System Architectures for Tightly-coupled Multiprocessors", Michel Dubois and Faye A. Briggs © 1990.

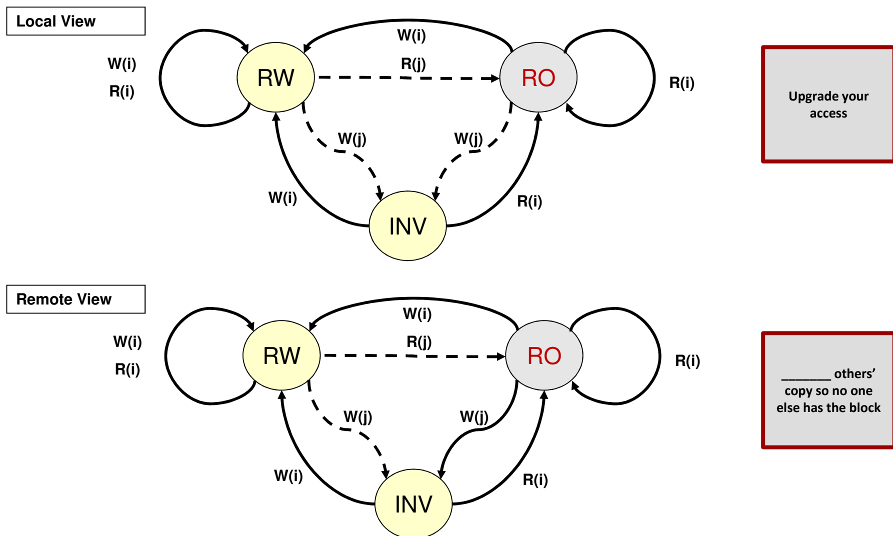
Remote Read



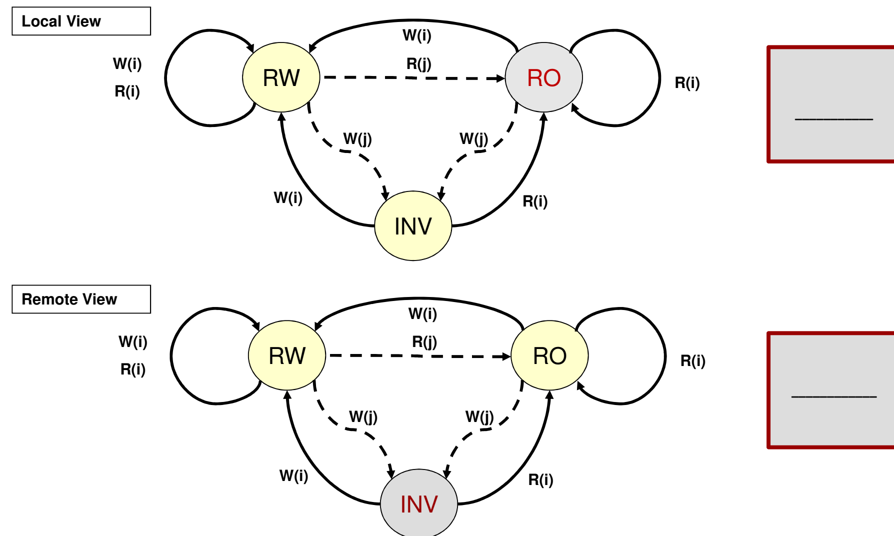
If you have the only couple and another processor wants to read the data

The other processor goes from ___ to ___

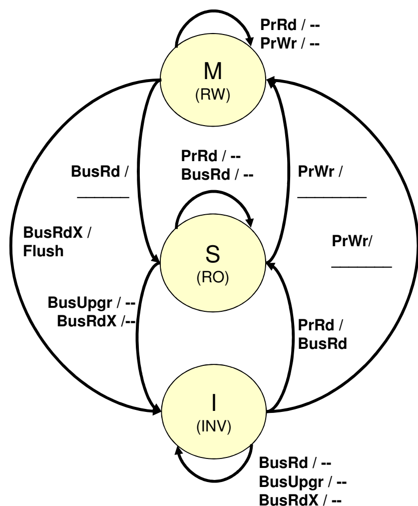
Local Write



Remote Read



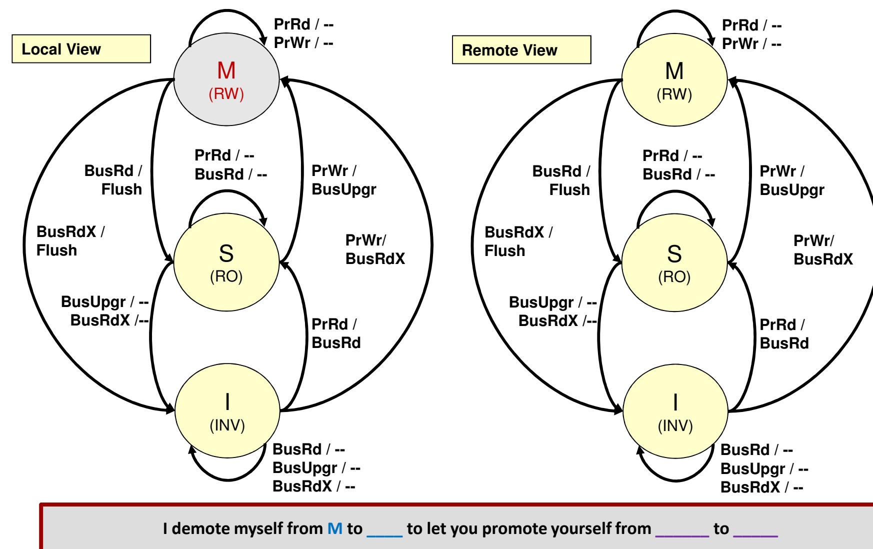
Write Invalidate Snoopy Protocol



Acronyms	Description
PrRd	Processor Read
PrWr	Processor Write
BusRd	Read request for a block
BusWrite	Write a word to memory and invalidate other copies
BusUpgr	Invalid other copies
BusUpdate	Update other copies
BusRdX	Read block and invalidate other copies
Flush	Supply a block to a requesting cache
S	Shared line is activated
~S	Shared line is deactivated

Michel Dubois, Murali Annavaram and Per Stenström © 2011.

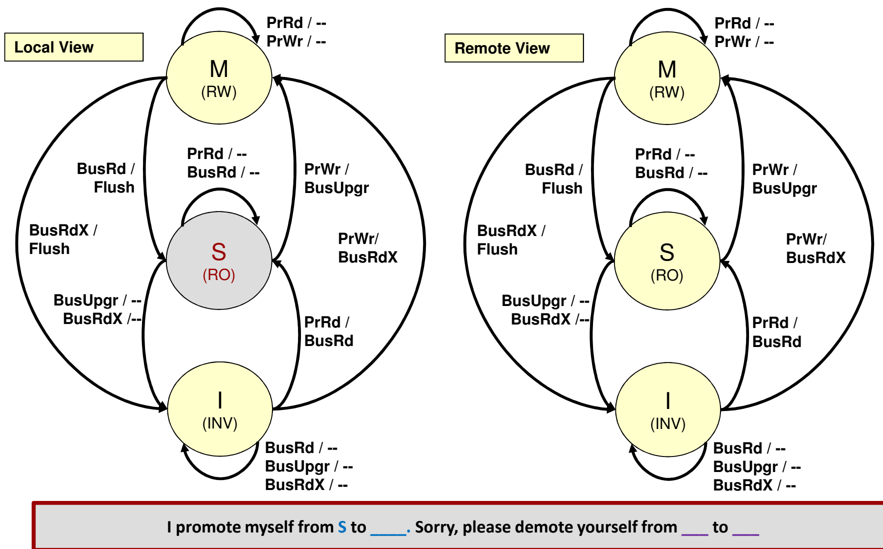
Remote Read



I demote myself from M to ____ to let you promote yourself from ______ to _____

Michel Dubois, Murali Annavaram and Per Stenström © 2011.

Local Write



Michel Dubois, Murali Annavaram and Per Stenström © 2011.

Write Invalid Snoopy Protocol

- Read miss:
  - If the block is not present in any other cache, or if it is present as a Shared copy, then the _____________ and all copies remain ________
  - If the block is present in a different cache in Modified state, then that cache ____________________ and ______________ at the same time; both copies become ________
- Read Hit
  - No action is taken

Write Invalid Snoopy Protocol

- Write hit:
  - If the local copy is Modified then no action is taken
  - If the local copy is Shared, then an ____________ must be sent to all processors which have a copy

Write Invalid Snoopy Protocol

- Write miss:
  - If the block is Shared in other cache or not present in other caches, memory responds in both cases, and in the first case all ________ copies are ______________
  - If the block is Modified in another cache, that cache responds, then _________ its copy
- Replacement
  - If the block is Modified, then _________ must be updated

Coherency Example

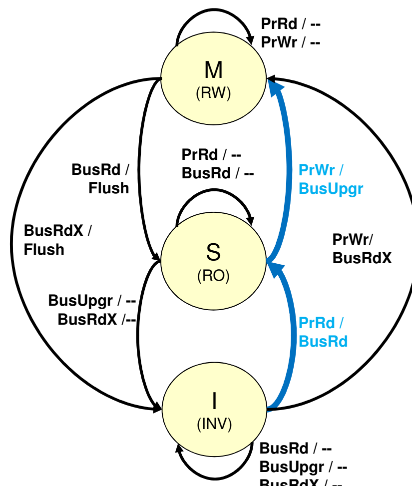
Processor Activity	Bus Activity	P1 $ Content	P1 Block State (M,S,I)	P2 $ Content	P2 Block State (M,S,I)	Memory Contents
		-	-	-	-	A
P1 reads block X	BusRd					
P2 reads block X	BusRd					
P1 writes block X=B						
P2 reads block X						

Updated Coherency Example

Processor Activity	Bus Activity	P1 $ Content	P1 Block State (M,S,I)	P2 $ Content	P2 Block State (M,S,I)	Memory Contents
		-	-	-	-	A
P1 reads block X	BusRd					
P1 writes X=B						
P2 writes X=C						
P1 reads block X						

Problem with MSI

- Read miss followed by write causes two bus accesses
- Solution: MESI
  - New "Exclusive" state that indicates you have the ______ copy and can ________ modify it

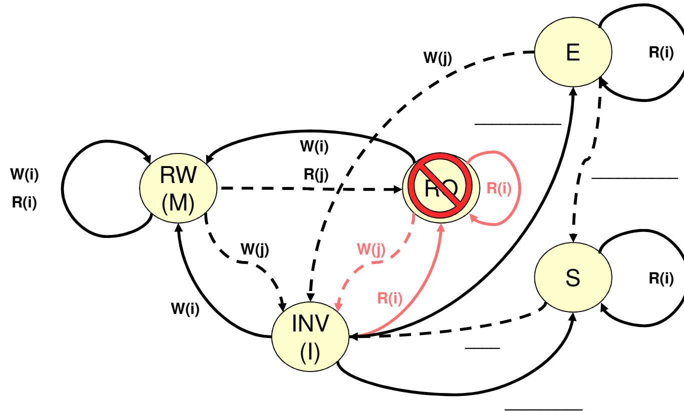


Exclusive State & Shared Signal

- Exclusive state avoid need to perform BusUpgr when moving from Shared to Modified even when no other copy exists
- New state definitions:
  - Exclusive = only copy of (**modified / unmodified**) cache block
  - Shared = multiple copies exist of (**modified / unmodified**) cache block
- New "Shared" handshake signal is introduced on the bus
  - When a read request is placed on the bus, other snooping caches assert this signal if they have a copy
  - If signal is not asserted, the reader can assume __________ access

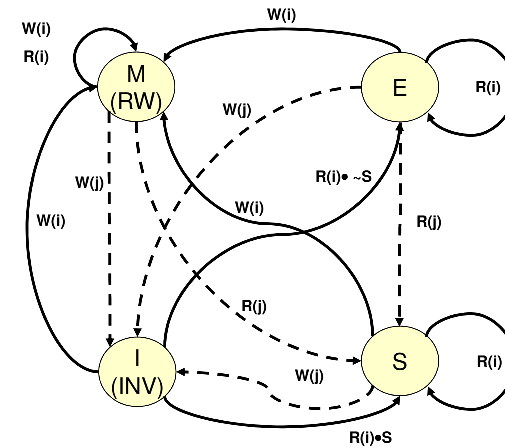
Updated MESI Protocol

- Convert RO to two states: Shared & Exclusive



Updated MESI Protocol

- Final Resulting Protocol



MESI

Processor Activity	Bus Activity	P1 $ Content	P1 Block State (MESI)	P2 Block State (MESI)	P3 Block State (MESI)	Memory Contents
		-	-	-	-	A
P1 reads block X						
P1 writes X=B						
P2 reads X						
P3 reads block X						

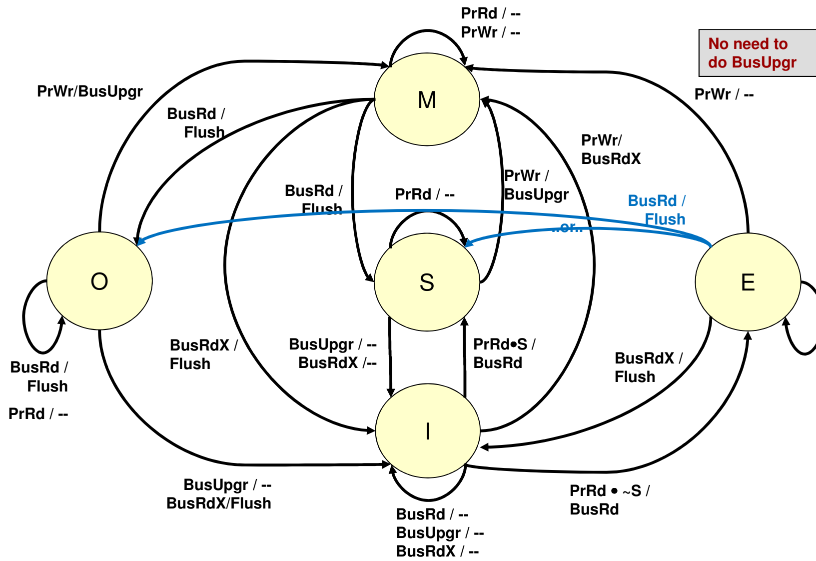
When P3 reads and the block is in the shared state, the slow memory supplies the data.

We can add an "Owned" state where one cache takes "ownership" of a shared block and supplies it quickly to other readers when they request it. The result is MOESI.

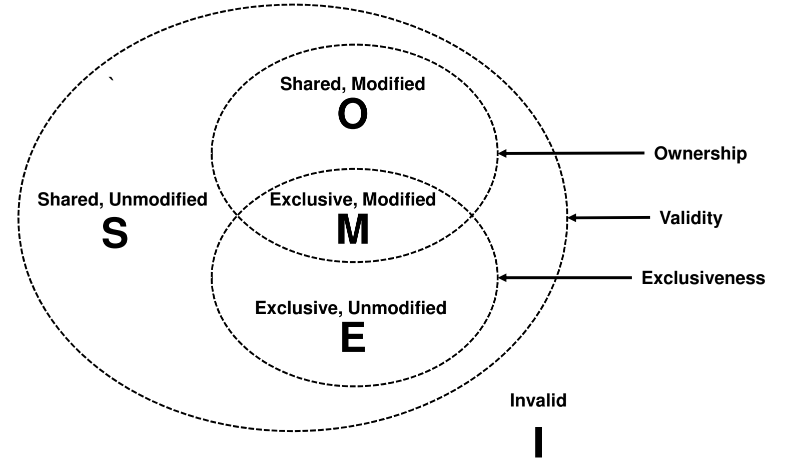
Owned State

- In original MSI, lowering from M to S or I causes a flush of the block
  - This also causes an updating of main memory which is slow
- It is best to postpone updating main memory until absolutely necessary
  - The M=>S transition is replaced by M=>O
  - Main memory is left in the stale state until the Owner needs to be invalidated in which case it is flushed to main memory
  - In the interim, any other cache read request is serviced by the owner quickly
- Summary: Owner is responsible for…
  - Supplying a copy of the block when another cache requests it
  - Transferring ownership back to main memory when it is invalidated

MOESI

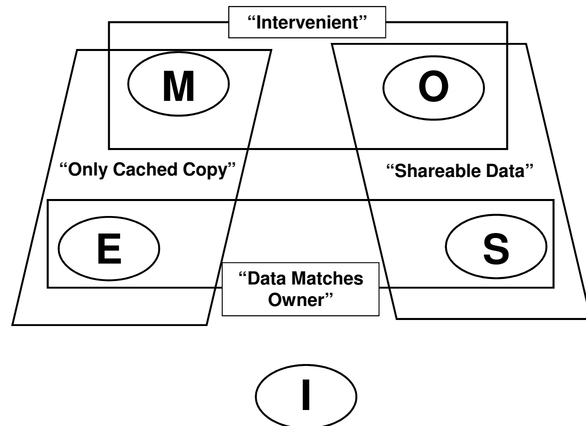


Characteristics of Cached Data



A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, P. Sweazy and A. J. Smith © 1986.

MOESI State Pairs



A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, P. Sweazy and A. J. Smith © 1986.