CHAPTER 5

MULTIPROCESSOR SYSTEMS

- PARALLEL PROGRAMMING MODELS
- MESSAGE-PASSING SYSTEMS
- SHARED-MEMORY ARCHITECTURES
- COHERENCE PROTOCOLS—BUS SNOOPING
- COHERENCE PROTOCOLS—DIRECTORIES: CC-NUMA

PARALLEL PROGRAMMING MODELS

- **HOW PARALLEL COMPUTATIONS CAN BE EXPRESSED IN A HIGH-LEVEL LANGUAGE**
  - SIMPLE EXTENSIONS THROUGH AN API (APPLICATION-PROGRAMMING INTERFACE) OF COMMON PROGRAMMING LAGUAGES SUCH AS C/C++ OR FORTRAN
    - SHARED-MEMORY: OPEN-MP OR PTHREADS
    - MESSAGE-PASSING: MPI (MESSAGE-PASSING INTERFACE)
  - PARALLELIZING COMPILERS TRANSLATE A SEQUENTIAL PROGRAM INTO PARALLEL THREADS
  - THREADS OR PROCESSES MUST COMMUNICATE AND COORDINATE THEIR ACTIVITY (SYNCHRONIZATION)

- **CONSIDER TWO PROGRAM SEGMENTS S1 AND S2 SO THAT S1 PRECEDES S2 IN THE SEQUENTIAL CODE**
  - TO RUN IN PARALLEL S1 MUST AVOID WRITING INTO VARIABLES THAT ARE INPUT TO S2 (RAW HAZARD)
  - AND, S2 MUST AVOID WRITING INTO VARIABLES THAT ARE WRITTEN OR READ BY S1 (WAW AND WAR HAZARDS)
  - TO CONFORM TO SEQUENTIAL SEMANTICS

DATA vs FUNCTION PARALLELISM

- **DATA PARALLELISM:**
  - PARTITION THE DATA SET
  - APPLY THE SAME FUNCTION TO EACH PARTITION
  - SPMD (SINGLE PROGRAM MULTIPLE DATA)
    - (NOT SIMD)
  - MASSIVE PARALLELISM

- **FUNCTION PARALLELISM**
  - INDEPENDENT FUNCTIONS ARE EXECUTED ON DIFFERENT PROCESSORS
  - E.G.: STREAMING--SOFTWARE PIPELINE
  - MORE COMPLEX, LIMITED PARALLELISM

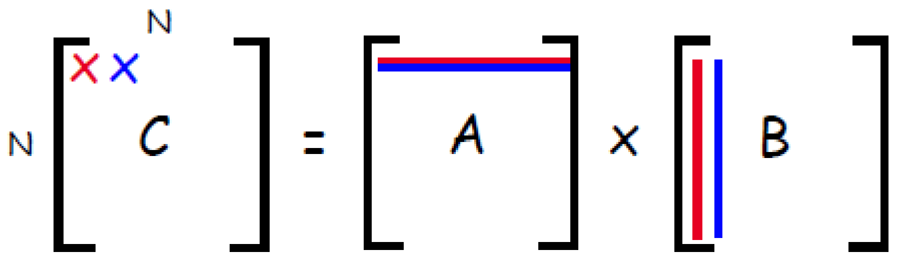
- **COMBINE BOTH: FUNCTION + DATA PARALLELISM**

EXAMPLE: MATRIX MULTIPLY + SUM

SEQUENTIAL PROGRAM:

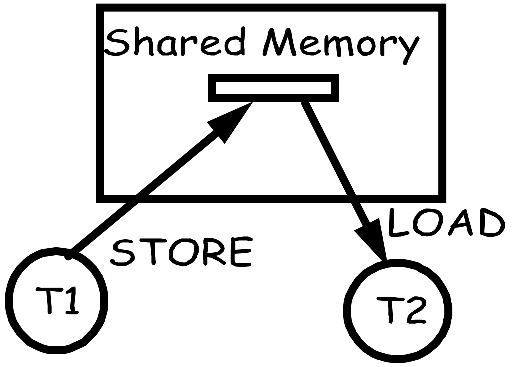
```
sum = 0;
for (i=0,i<N, i++)
        for (j=0,j<N, j++){
            C[i,j] = 0;
            for (k=0,k<N, k++)
                    C[i,j] = C[i,j] + A[i,k]*B[k,j];
                sum += C[i,j];
        }
```

- MULTIPLY MATRICES A[N,N] BY B[N,N] AND STORE RESULT IN C[N,N]
- ADD ALL ELEMENTS OF C



INTER-PE COMMUNICATION

- **IMPLICITELY VIA MEMORY**



  - PROCESSORS SHARE SOME MEMORY
  - COMMUNICATION IS IMPLICIT THROUGH LOADS AND STORES
    - NEED TO SYNCHRONIZE
    - NEED TO KNOW HOW THE HARDWARE INTERLEAVES ACCESSES FROM DIFFERENT PROCESSORS

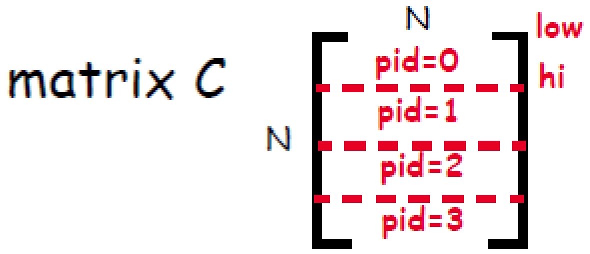
- **EXPLICITELY VIA MESSAGES (SENDS AND RECEIVES)**
  - NEED TO KNOW THE DESTINATION AND WHAT TO SEND
  - EXPLICIT MESSAGE-PASSING STATEMENTS IN THE CODE
  - CALLED "MESSAGE PASSING"

NO HYPOTHESIS ON THE RELATIVE SPEED OF PROCESSORS

EXAMPLE: MATRIX MULTIPLY + SUM

SHARED-MEMORY PROGRAM

```
    /* A, B, C, BAR, LV and sum are shared
    /* All other variables are private
1a            low = pid*N/nproc;                  /pid=0...nproc-1
1b            hi = low + N/nproc;                      /rows of A
1c        mysum = 0; sum = 0;  /A and B are in
2         for (i=low,i<hi, i++)     /shared memory
3                 for (j=0,j<N, j++){
4                         C[i,j] = 0;
5                         for (k=0,k<N, k++)
6                             C[i,j] = C[i,j] +  A[i,k]*B[k,j]; /at the end matrix
    C is
7                         mysum +=C[i,j];                /C is in shared memory
8                     }
9         BARRIER(BAR);
10        LOCK(LV);
11                sum += mysum;
12         UNLOCK(LV);
```



MUTUAL EXCLUSION

- Need for "Mutual Exclusion"
- Assume the following statements are executed by 2 threads, T1 and T2, on Sum

```
T1                       T2
Sum<- Sum+1                  Sum<- Sum+1
```

  - The programmer's expectation is that, whatever the order of execution of the two statements is, the final result will be that Sum is incremented by 2
  - However program statements are not executed in an atomic fashion.

- Compiled code on a RISC machine includes several instructions
- A possible interleaving of execution is:

```
T1                       T2
r1 <- Sum
                         r1 <- Sum
r1 <- r1 + 1
                         r1 <- r1 + 1
Sum <- r1
                         Sum <- r1
```

- At the end the result is that Sum is incremented by 1 (NOT 2)

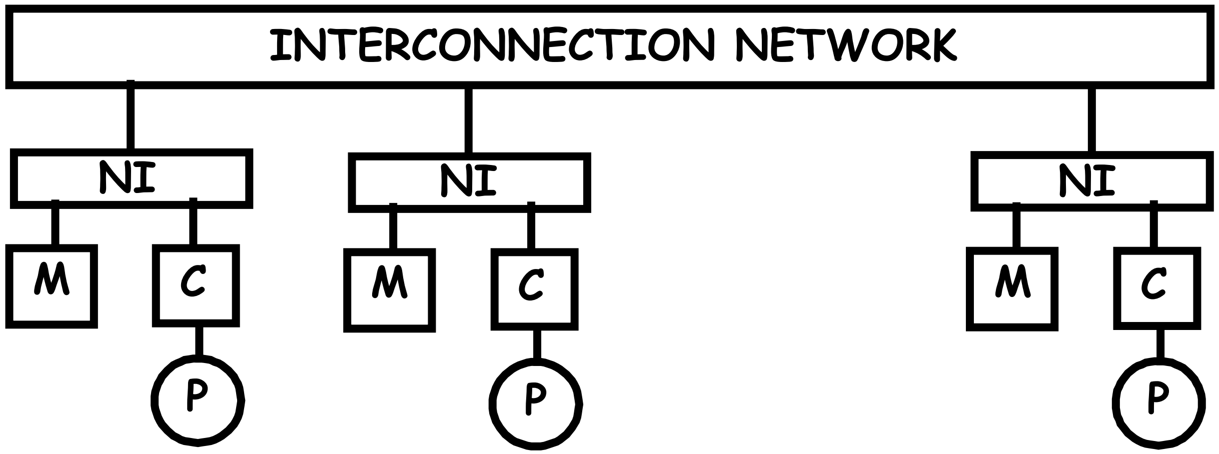
EXAMPLE: MATRIX MULTIPLY + SUM

MESSAGE-PASSING PROGRAM

```
1a myN = N/nproc;
1b if(pid == 0){
1c   for(i=1; i<nproc;i++){
1d     k=i*N/nproc;
1g     SEND(&A[k][0],myN*N*sizeof(float),i,IN1);
1h     SEND(&B[0][0],N*N*sizeof(float),i,IN2);
1i   } else {
1j   RECV(&A[0][0],myN*N*sizeof(float),0,IN1);
1k   RECV(&B[0][0],N*N*sizeof(float),0,IN2);
1l }
1k mysum = 0;
2  for (i=0,i<myN, i++)
3   for (j=0,j<N, j++){
4     C[i,j] = 0;
5     for (k=0,k<N, k++)
6     C[i,j] = C[i,j] + A[i,k]*B[k,j];
7     mysum += C[i,j];
8   }
```

```
9  if (pid == 0){
10     sum = mysum;
11     for(i=1;i<nproc;i++){
12    RECV(&mysum,sizeof(float),i,SUM);
13       sum +=mysum;
14     }
15     for(i=1; i<nproc;i++){
         k=i*N/nproc;
16    RECV(&C[k][0],myN*N*sizeof(float),i,RES)
20     }
21 } else{
22     SEND(&mysum,sizeof(float),0,SUM);
23 SEND(&C[0][0],myN*N*sizeof(float),0,RES);
24 }
```

MESSAGE-PASSING: MULTICOMPUTERS



- PROCESSING NODES INTERCONNECTED BY A NETWORK
- COMMUNICATION CARRIED OUT BY MESSAGE EXCHANGES
- SCALES WELL
- HARDWARE IS INEXPENSIVE
- SOFTWARE IS COMPLEX

SYNCHRONOUS MESSAGE-PASSING

- **CODE FOR THREAD T1:**

```
A = 10;
SEND(&A,sizeof(A),T2,SEND_A);
A = A+1;
RECV(&C,sizeof(C),T2,SEND_B);
printf(C);
```

CODE FOR THREAD T2:

```
B = 5;
RECV(&B,sizeof(B),T1,SEND_A);
B=B+1;
SEND(&B,sizeof(B),T1,SEND_B);
```

- **EACH SEND/RECV HAS 4 OPERANDS:**
  - STARTING ADDRESS IN MEMORY
  - SIZE OF MESSAGE
  - DESTINATION/SOURCE THREAD ID
  - TAG CONNECTING SENDS AND RECEIVES

- **IN SYNCHRONOUS M-P SENDER BLOCKS UNTIL RECV IS COMPLETED AND RECEIVER BLOCKS UNTIL MESSAGE HAS BEEN SENT**
  - NOTE: THIS IS MUCH MORE THAN WAITING FOR MESSAGE PROPAGATION

- **QUESTION: WHAT IS THE VALUE PRINTED UNDER SYNCHRONOUS M-P?**
  - VALUE 10 IS RECEIVED IN B BY T2; B IS INCREMENTED BY 1
  - THEN THE NEW VALUE OF B (11) IS SENT AND RECEIVED BY T1 INTO C
  - AND THREAD 1 PRINTS “11”

SYNCHRONOUS MESSAGE-PASSING

- **ADVANTAGE: ENFORCES SYNCHRONIZATION**
- **DISADVANTAGES:**
  - PRONE TO DEADLOCK
  - BLOCK THREADS (NO OVERLAP OF COMMUNICATION WITH COMPUTATION)

- **DEADLOCK EXAMPLE:**

CODE FOR THREAD T1:

```
A = 10;
SEND(&A,sizeof(A),T2,SEND_A);
RECV(&C,sizeof(C),T2,SEND_B);
```

CODE FOR THREAD T2:

```
B = 5;
SEND(&B,sizeof(B),T1,SEND_B)
RECV(&D,sizeof(D),T1,SEND_A);
```

TO ELIMINATE THE DEADLOCK: SWAP THE SEND/RECV PAIR IN T2 OR EMPLOY ASYNCHRONOUS MESSAGE-PASSING

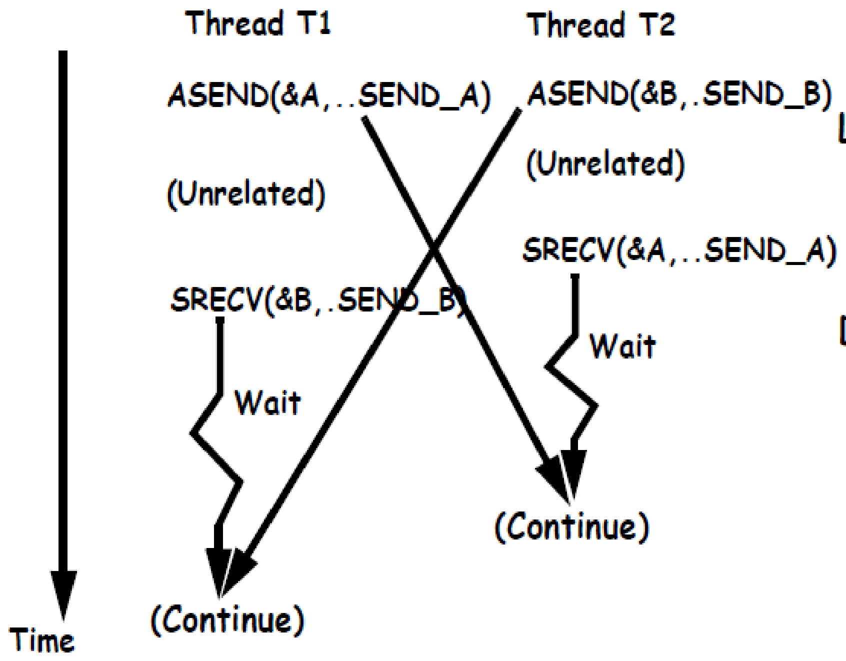
SYNCHRONOUS MESSAGE-PASSING

CODE FOR THREAD T1:

```
A = 10;
ASEND(&A,sizeof(A),T2,SEND_A);
  <Unrelated computation;>
SRECV(&B,sizeof(B),T2,SEND_B);
```

CODE FOR THREAD T2:

```
B = 5;
ASEND(&B,sizeof(B),T1,SEND_B);
  <Unrelated computation;>
SRECV(&A,sizeof(B),T1,SEND_A);
```



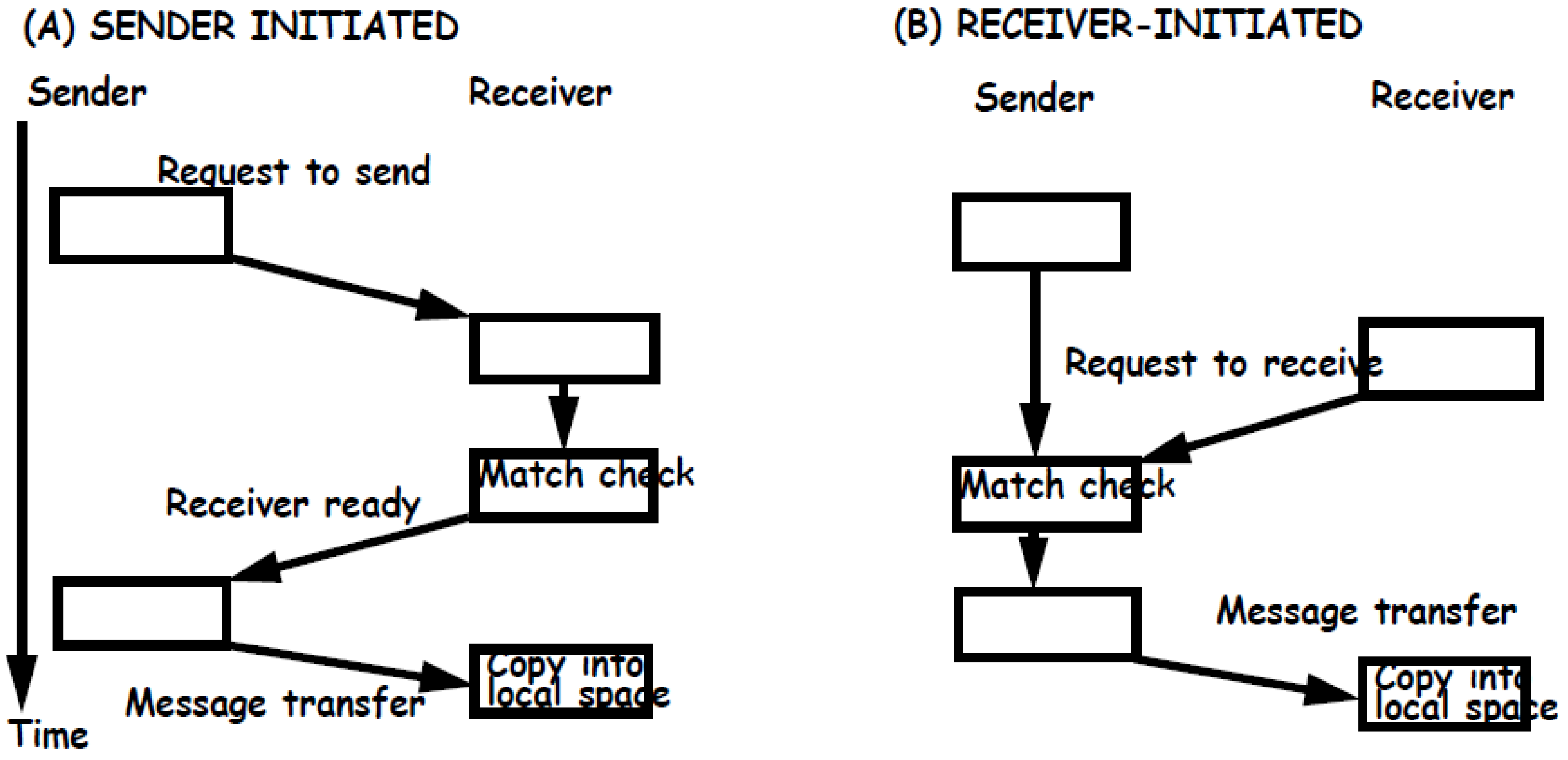
LOCAL COMPUTATION IS OVERLAPPED WITH SENDING

DEADLOCK IS BROKEN BECAUSE THREADS DON'T WAIT AT SEND

- **BLOCKING vs NON-BLOCKING MESSAGE PASSING**
  - BLOCKING: RESUME ONLY WHEN AREA OF MEMORY CAN BE MODIFIED
  - NON-BLOCKING: RESUME EARLY--RELIES ON PROBE MESSAGES

SYNCHRONOUS MESSAGE-PASSING

IN GENERAL MULTIPLE THREADS ARE RUNNING ON A CORE

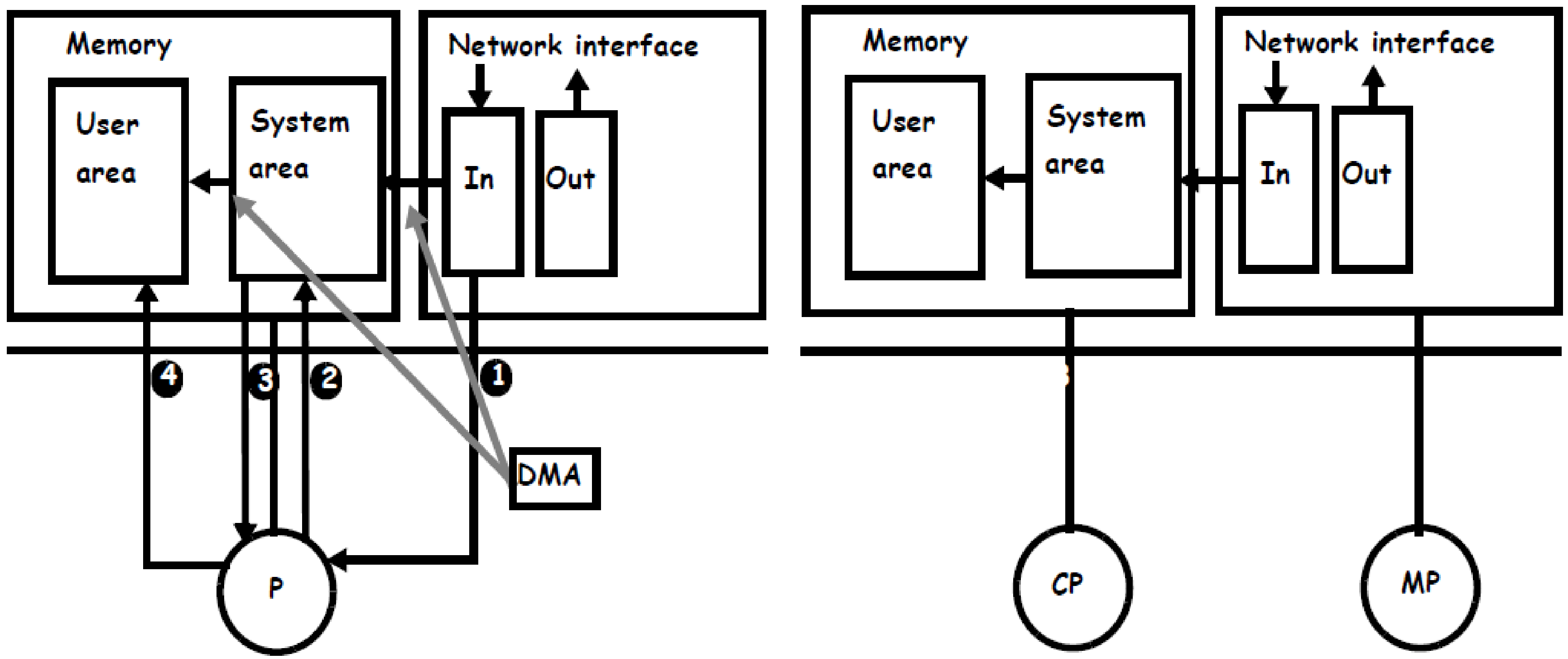


- THIS HANDSHAKE IS FOR SYNCHRONOUS M-P
- CAN BE APPLIED TO BLOCKING OR NON-BLOCKING ASYNCHRONOUS
  - SENDER CONTINUES AFTER SEND (A) AND RECVer AFTER RECV (B)
    - IF BLOCKING, SENDER MUST MAKE A COPY OF MESSAGE FIRST

SUPPORT FOR MESSAGE PASSING PROTOCOLS

- **DATA MUST BE COPIED FROM/TO MEMORY TO/FROM NI**
  - PROCESSOR COULD DO IT
  - DMA (DIRECT MEMORY ACCESS) CAN SPEED UP MESSAGE TRANSFERS AND OFF-LOAD THE PROCESSOR
  - DMA IS PROGRAMMED BY THE PROCESSOR;
    - START ADDRESS AND SIZE
- **DEDICATED MESSAGE PROCESSORS**
  - USE A SPECIAL PROCESSOR TO PROCESS MESSAGES ON BOTH ENDS
  - THUS RELIEVING THE COMPUTE O/S FROM DOING IT
- **SUPPORT FOR USER-LEVEL MESSAGES**
  - BASIC MESSAGE PASSING SYSTEMS DRIVE DMA ENGINE FROM O/S
  - THIS IS NEEDED FOR PROTECTION BETWEEN USERS
  - MESSAGE IS FIRST COPIED INTO SYSTEM SPACE AND THEN INTO USER SPACE (RECEIVE)
  - MESSAGE IS COPIED FROM USER SPACE TO SYSTEM SPACE (SEND)
  - TAG USER MESSAGES SO THAT THEY ARE PICKED UP AND DELIVERED DIRECTLY IN USER MEMORY SPACE (ZERO COPY)

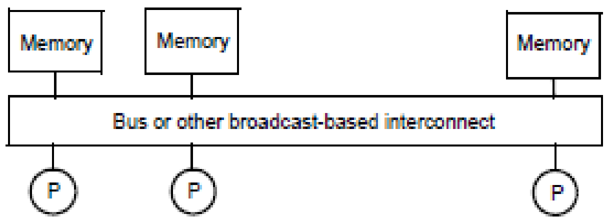
MESSAGE-PASSING SUPPORT



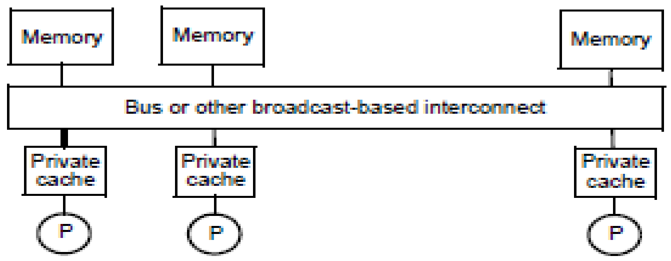
BUS-BASED SHARED MEMORY SYSTEMS

- **MULTIPROCESSOR CACHE ORGANIZATIONS**
- **SNOOPY CACHE PROTOCOLS**
  - A SIMPLE SNOOPY CACHE PROTOCOL
  - DESIGN SPACE OF SNOOPY CACHE PROTOCOLS
  - PROTOCOL VARIATIONS
  - MULTI-PHASE CACHE PROTOCOLS
- **CLASSIFICATION OF COMMUNICATION EVENTS**
- **TRANSLATION LOOKASIDE BUFFER (TLB) CONSISTENCY**

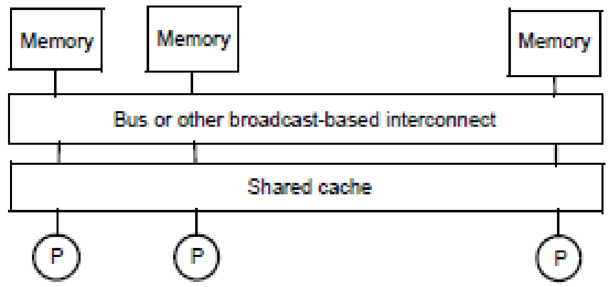
CACHE ORGANIZATIONS



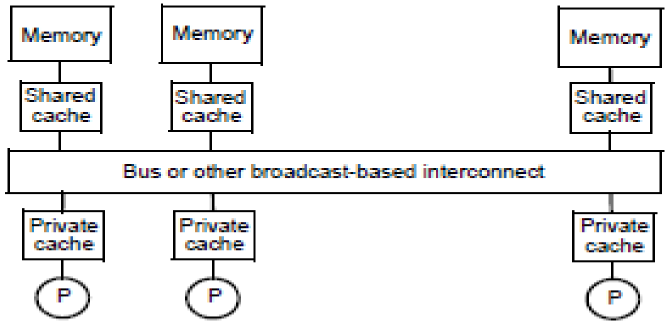
(a) Dance-hall multiprocessor architecture or SMP



(c) SMP with private caches



(b) SMP with shared level 1 cache



(d) SMP with private caches and shared Level 2 cache

(a) CONCEPTUAL – NOT PRACTICAL; NO CACHE

(b) SHARED L1 CACHE; NO REPLICATION BUT HIGHER HIT LATENCY THAN IN (c)

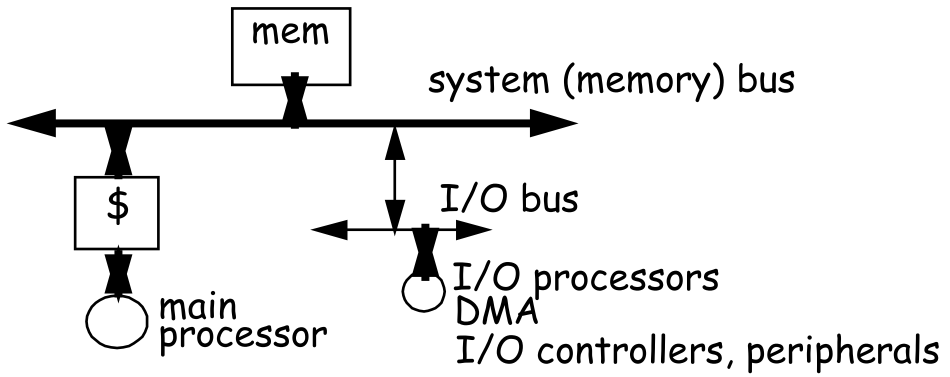
(c) PRIVATE L1 CACHES

(d) PRIVATE L1 CACHES AND SHARED L2; POPULAR FOR CHIP MULTIPROCESSORS (CMPS)

REST OF DISCUSSION CONSIDERS ORGANIZATION (c) AND (d)

COHERENCE

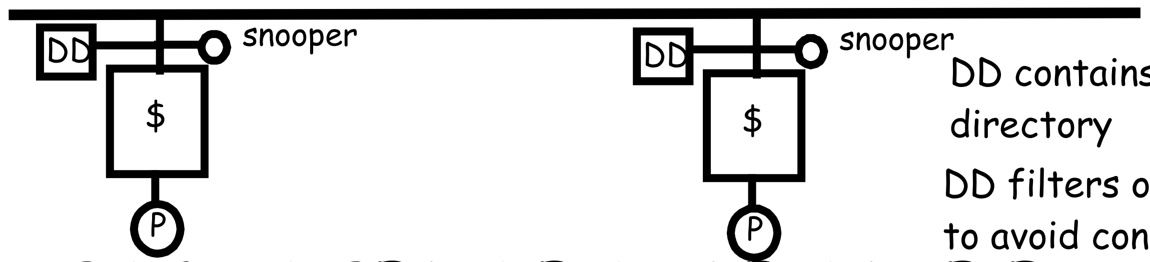
- **IN UNIPROCESSORS, A LOAD MUST RETURN THE VALUE OF THE LATEST STORE IN PROCESS ORDER WITH THE SAME ADDRESS**
  - THIS IS DONE THROUGH MEMORY DISAMBIGUATION AND MANAGMENT OF CACHE HIERARCHY
  - SOME PROBLEMS WITH I/O, AS I/O IS OFTEN CONNECTED TO MEMORY BUS



- **COHERENCE BETWEEN I/O TRAFFIC AND CACHE MUST BE ENFORCED**
  - HOWEVER, THIS IS INFREQUENT AND SOFTWARE IS INFORMED
  - SO SOFTWARE SOLUTIONS WORK
  - UNCACHEABLE MEMORY, UNCACHEABLE OPS, CACHE FLUSHING
  - ANOTHER SOLUTION IS TO PASS I/O THROUGH CACHE
- **IN MULTIPROCESSORS THE COHERENCE PROBLEM IS PERVASIVE, PERFORMANCE CRITICAL AND SOFTWARE IS NOT INFORMED**
  - SHARING OF DATA, THREAD MIGRATION AND I/O
  - COMMUNICATION IS IMPLICIT
  - THUS HARDWARE MUST SOLVE THE PROBLEM.

SNOOPING-BASED CACHE COHERENCE

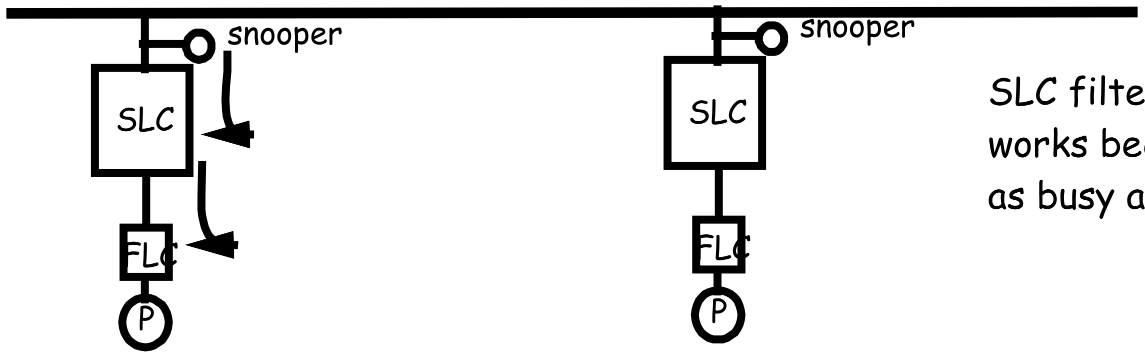
- **BASIC IDEA**
  - TRANSACTIONS ON THE BUS ARE "VISIBLE" TO ALL PROCESSORS
  - BUS INTERFACE CAN "SNOOP" (MONITOR) THE BUS TRAFFIC AND TAKE ACTION WHEN REQUIRED
  - TO TAKE ACTION THE "SNOOPER" MUST CHECK THE STATUS OF THE CACHE
- **DUAL DIRECTORY**



DD contains a copy of the cache directory

DD filters out bus request to avoid conflicts with the CPU

- **SNOOPING CAN BE DONE IN THE 2ND LEVEL CACHE**
  - SINCE THERE IS INCLUSION, SLC FILTERS OUT TRANSACTIONS TO FLC
  - SLC CONTAINS A BIT INDICATING WHETHER THE BLOCK IS IN FLC
  -

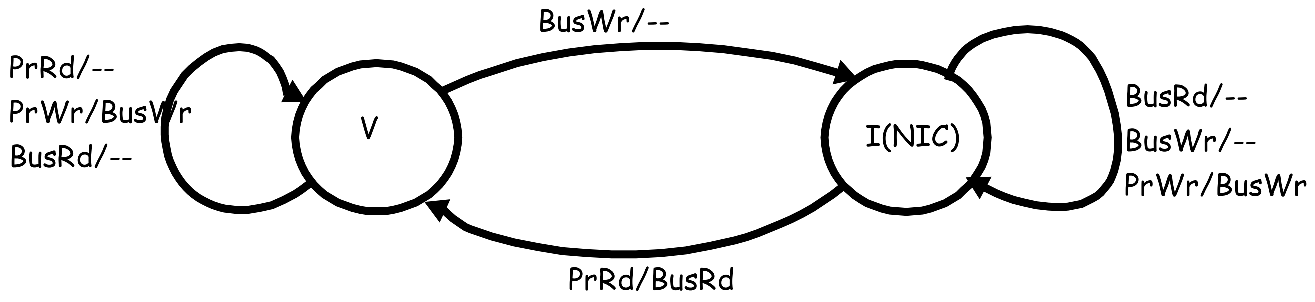


SLC filters out bus request works because SLC is not as busy as FLC

A SIMPLE PROTOCOL FOR WRITE-THROUGH CACHES

- **TO SIMPLIFY ASSUME NO ALLOCATE ON STORE MISSES**
  - ALL STOREs AND LOAD MISSES PROPAGATE ON THE BUS
- **STOREs MAY UPDATE OR INVALIDATE CACHES**

- **STATE DIAGRAM**
  - EACH CACHE IS REPRESENTED BY A FINITE STATE MACHINE
  - IMAGINE P IDENTICAL FSMs WORKING TOGETHER, ONE PER CACHE
  - FSM REPRESENTS THE BEHAVIOR OF A CACHE W.R.T. A MEMORY BLOCK
    - NOT THE CACHE CONTROLLER

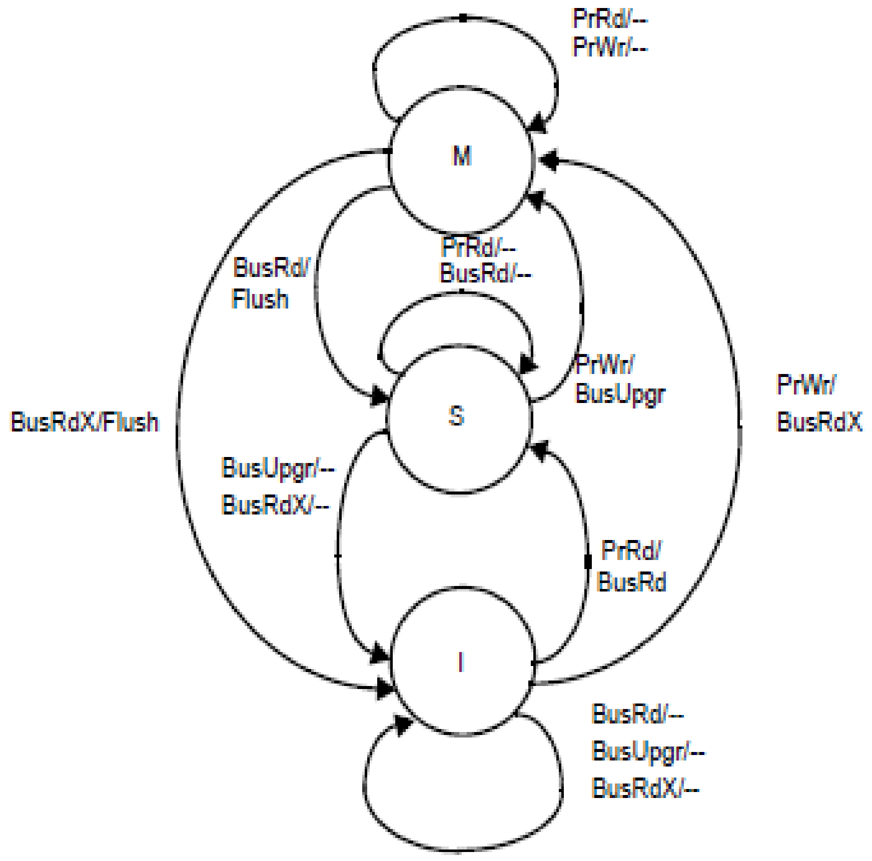


$$\begin{tikzcd}
V \arrow[loop left, "\text{PrRd/--, PrWr/BusWr, BusRd/--}"] \arrow[rr, bend left, "\text{BusWr/--}"] & & \text{I(NIC)} \arrow[ll, bend left, "\text{PrRd/BusRd}"] \arrow[loop right, "\text{BusRd/--, BusWr/--, PrWr/BusWr}"]
\end{tikzcd}$$

- NO NEW STATE (1 BIT PER BLOCK IN CACHE)

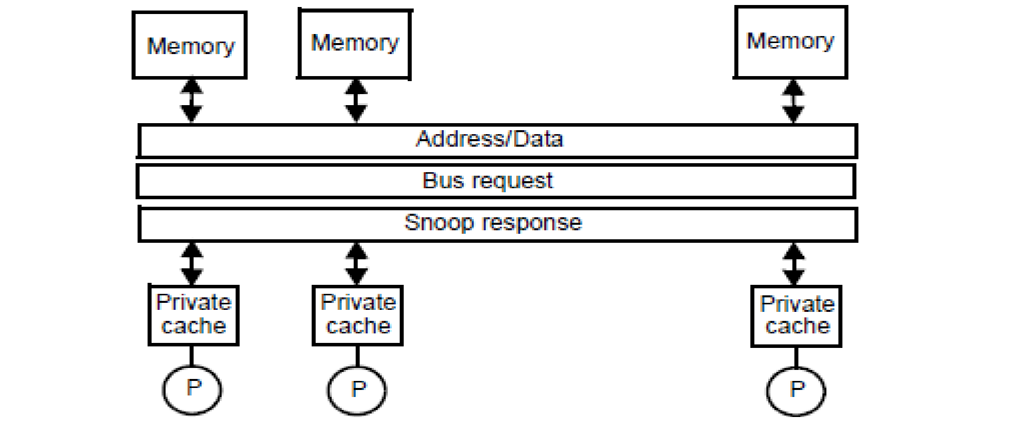
WRITE-BACK: MSI INVALIDATE PROTOCOL

- **BLOCK STATES:**
  - Invalid (I);
  - Shared (S): one copy or more, memory is clean;
  - Dirty (D) or Modified (M): one copy, memory is stale
- **PROCESSOR REQUESTS**
  - PrRd or PrWr
- **BUS TRANSACTIONS**
  - BusRd: requests copy with no intent to modify
  - BusRdX: requests copy with intent to modify
  - BusUpgr: invalidate remote copies

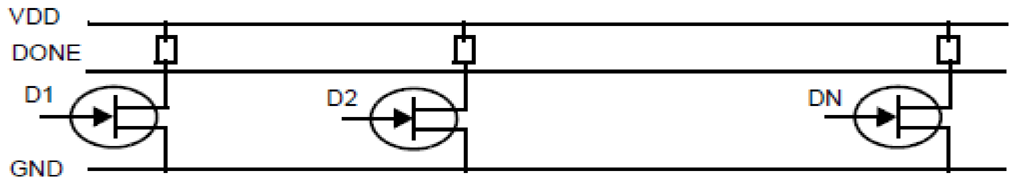


- WRITE TO SHARED BLOCK: USE BusUpgr
- FLUSH: FORWARD BLOCK COPY TO REQUESTER (MEMORY COPY IS STALE)
  - MEMORY SHOULD BE UPDATED AT THE SAME TIME

MSI: HARDWARE STRUCTURES



(a) Structure of a snooping bus

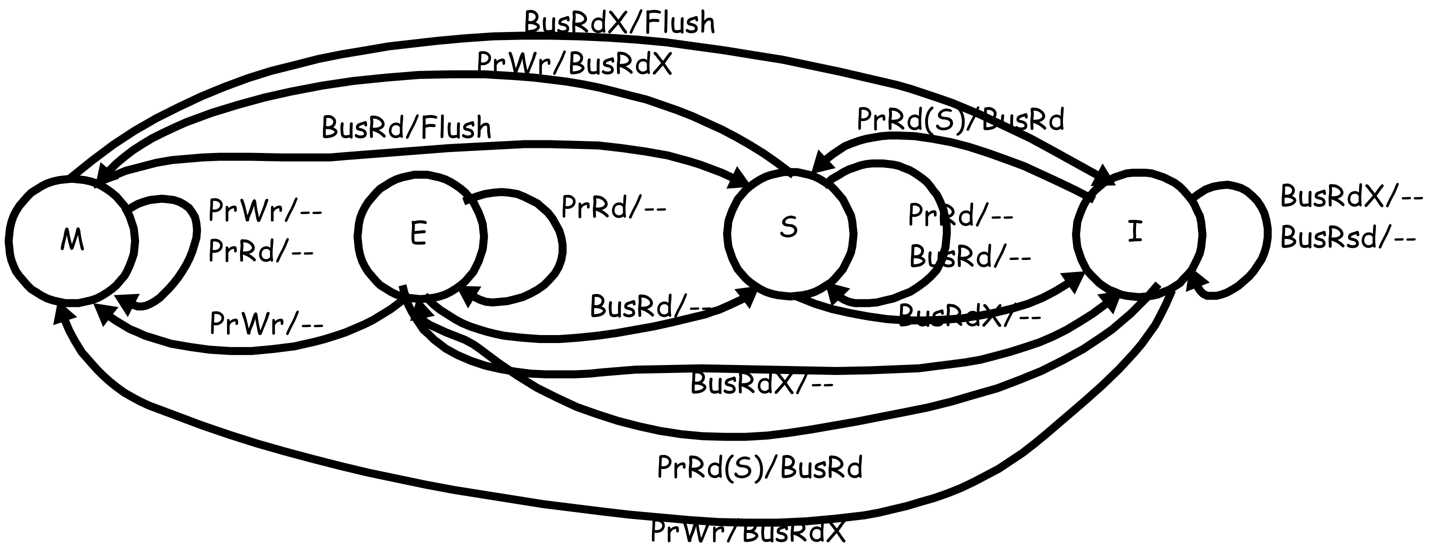


(b) Wired-NOR bus used in handshakes

- **A BUS TRANSACTION INVOLVES TRANSMITTING A BUS REQUEST ALONG WITH THE ADDRESS**
- **IN RESPONSE, THERE IS A SNOOP REPLY**
  - CAN BE ASYNCHRONOUS; USE OF HANDSHAKE ARRANGEMENT IN b)
  - CAN BE SYNCHRONOUS; MUST KEEP SNOOP RESPONSE LATENCY SHORT; USE OF DUAL DIRECTORIES

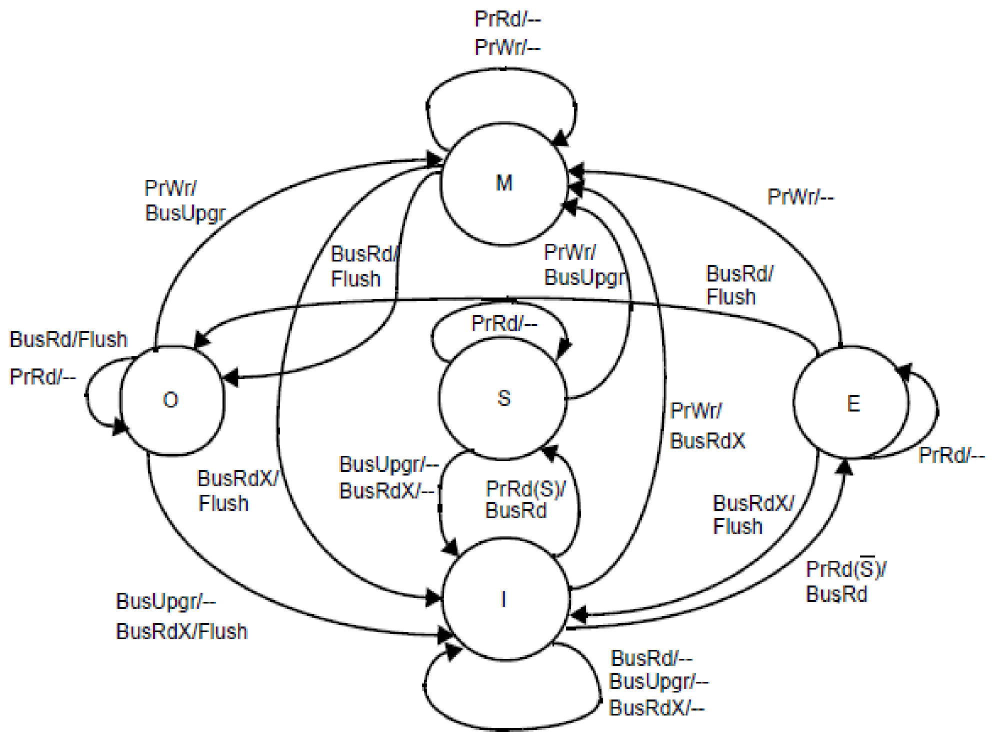
MESI PROTOCOL

- **PROBLEM WITH MSI INVALIDATE PROTOCOL**
  - MOST OF BLOCKS ARE PRIVATE--NOT SHARED
  - READ MISS FOLLOWED BY WRITE CAUSES TWO BUS ACCESSES
- **ADD AN EXCLUSIVE STATE**
  - Exclusive (E): one copy, memory is clean
  - use a Shared (S) line on bus to detect that the copy is unique



- On a read miss go to E or S depending on the value returned by the Shared line.
- Could use BusUpgr in transition from S to M

MOESI: A GENERAL CLASS OF PROTOCOLS



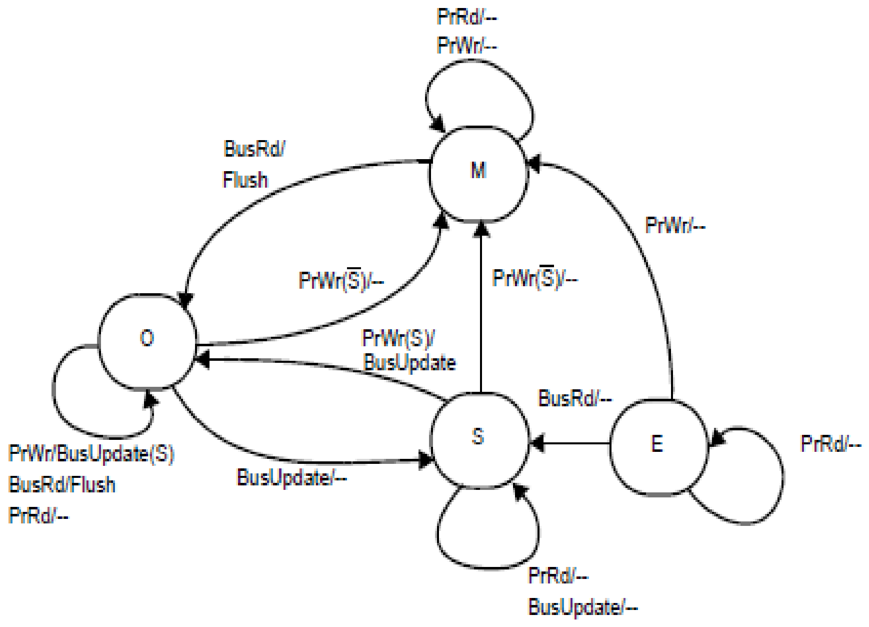
- MOESI CAN MODEL (AMONG OTHERS) MSI AND MESI
- ADDS NOTION OF OWNERSHIP TO AVOID UPDATING MEMORY ON EVERY WRITE; OWNER EVENTUALLY UPDATES MEMORY
- ALLOWS CACHE-TO-CACHE TRANSFERS BETWEEN OWNER AND REQUESTER
- OWNERSHIP IS TRANSFERRED TO ANOTHER CACHE OR TO MEMORY WHEN BLOCK IS INVALIDATED/REPLACED

CACHE PROTOCOL OPTIMIZATIONS

- **PRODUCER-CONSUMER SHARING; E.G. ONE PROCESSOR WRITES AND OTHERS (SAY N PROC.) READ THE SAME DATA**
  - RESULTS IN INVALIDATION FOLLOWED BY N READ MISSES
  - CAN DO BETTER BY LETTING ONE READ MISS BRING BLOCK INTO ALL N CACHES – READ SNARFING (OR READ BROADCAST).
- **MIGRATORY SHARING; PROCESSORS READ AND MODIFY SAME CACHE BLOCK – ONE AT A TIME**
  - COMMON BEHAVIOR IN DATA ACCESS INSIDE CRITICAL SECTIONS
  - RESULTS IN ONE COHERENCE MISS FOLLOWED BY AN INVALIDATION; THE INVALIDATION COULD BE SAVED.
  - OPTIMIZATION: ISSUE ONLY READ-EXCLUSIVE (COMBINED READ AND UPGRADE)
  - NEEDS SOME (SLIGHT) MODIFICATIONS TO A M(OE)SI PROTOCOL
- **UPDATE-BASED PROTOCOLS: UPDATE (INSTEAD OF INVALIDATE) BLOCK COPIES**
  - IF EVERY WRITTEN VALUE IS CONSUMED, UPDATE PROTOCOLS WILL OUTPERFORM INVALIDATE-BASED PROTOCOLS (SHORTER LATENCY AND LESS BANDWIDTH)

WRITE-BACK: MSI UPDATE PROTOCOL

- **BLOCK STATES:**
  - Invalid (I);
  - Shared (S): multiple copies, memory is clean;
  - Dirty (D) or Modified (M): one copy, memory is stale
- **PROCESSOR REQUESTS**
  - PrRd or Pr Wr
- **BUS TRANSACTIONS**
  - BusRd: requests copy
  - BusUpdate: update remote copies



- **SHARED BUS LINE (S): INDICATES WHETHER REMOTE COPIES EXIST**

TRADEOFF ANALYSIS: INVALIDATE vs UPDATE

WRITE-RUN OF AN ACCESS SEQUENCE TO THE SAME BLOCK: THE NUMBER OF WRITES IN A ROW BY A PROCESSOR BEFORE A READ OR WRITE IS ENCOUNTERED BY ANOTHER PROCESSOR

- Q: WHAT IS THE LENGTH OF THE WRITE RUN IN THE FOLLOWING SEQUENCE?

  R1, W1, R1, W1, W2, R2,..
- A: TWO BECAUSE AFTER THE 2ND WRITE BY P1, P2 WRITES

BANDWIDTH TRADEOFF (INVALIDATE vs UPDATE):

- WRITE-RUN OF LENGTH N
- INVALIDATE: BANDWITH = B(UPGRADE) + B(READ MISS)
- UPDATE: BANDWIDTH = N X B(UPDATE)
- ASSUME: B(UPGRADE) = B(UPDATE)

UPDATE OUTPERFORMS INVALIDATE WHEN

N < 1 + B(READ MISS)/B(UPDATE)

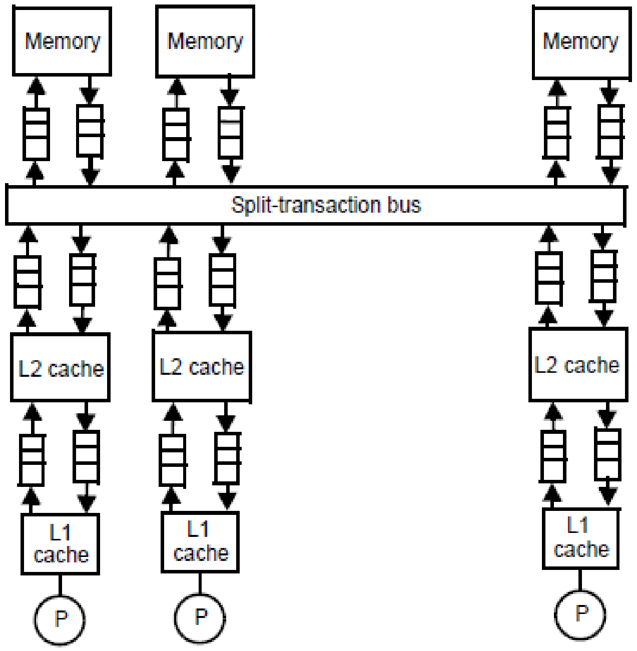
MULTIPHASE CACHE PROTOCOLS

MACHINE MODELS SO FAR HAVE ASSUMED

- SINGLE LEVEL OF PRIVATE CACHES
- "ATOMIC" (NON-PIPELINED) BUSES

A REALISTIC MODEL COMPRISES

- A MULTI-LEVEL (PRIVATE) CACHE HIERARCHY
- A SPLIT-TRANSACTION (PIPELINED) BUS



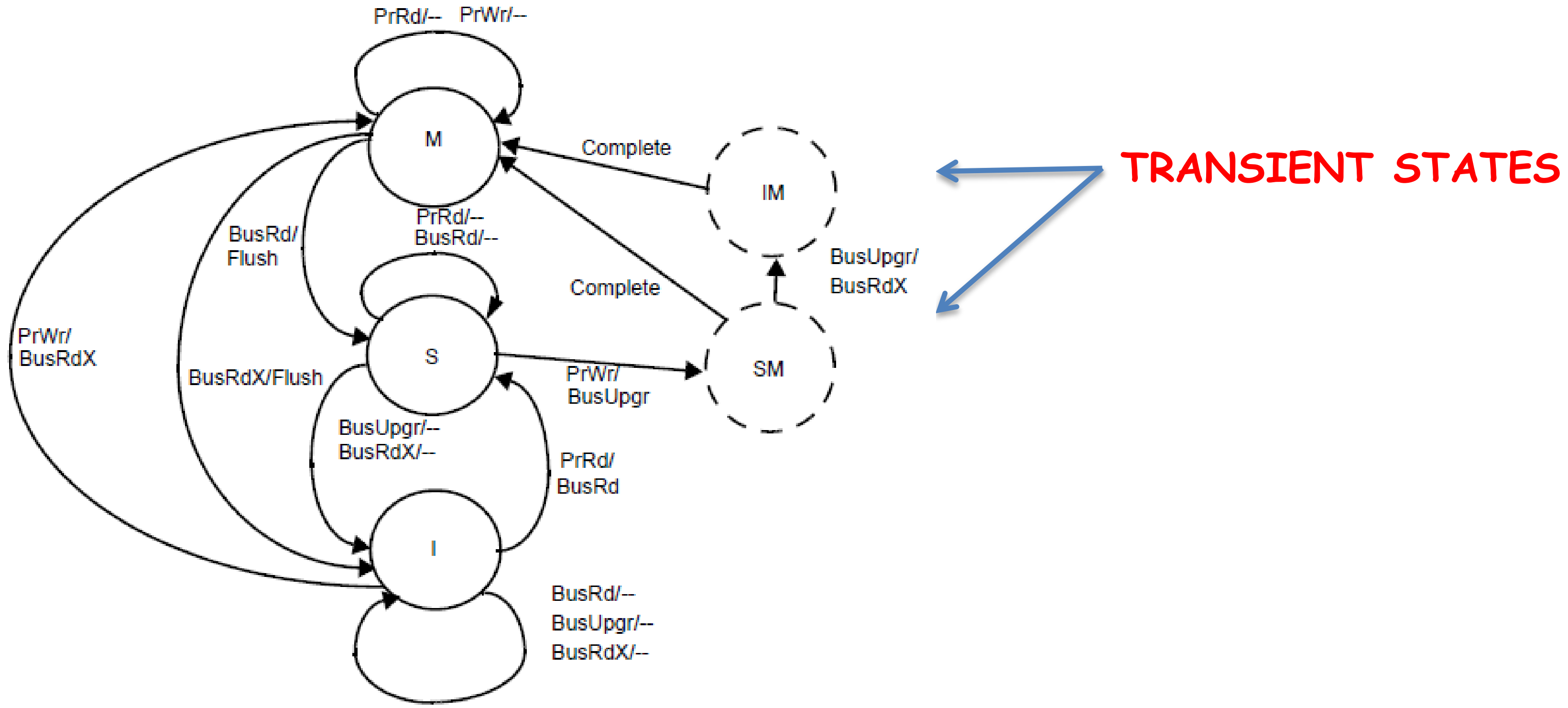
TRANSIENT (NON-ATOMIC) CACHE STATES

USING 'STABLE' STATES (E.G. M S I) AS AN ABSTRACTION IS TOO HIGH-LEVEL TO RESOLVE RACES IN CACHE PROTOCOLS

EXAMPLE)

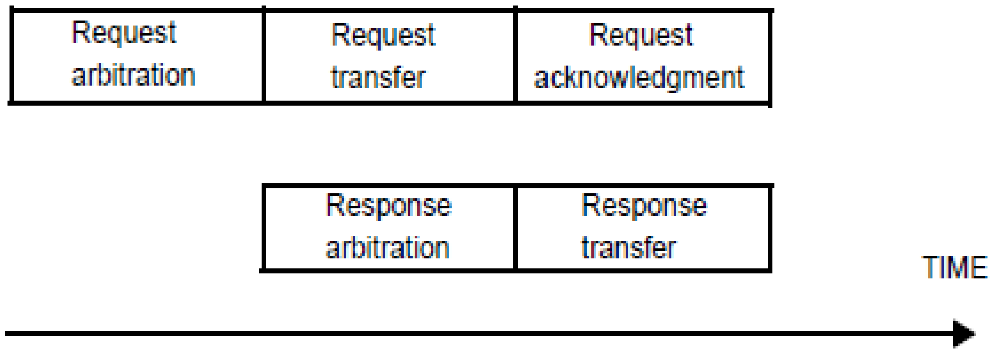
- Q: TWO PROCESSORS ISSUE A BUS UPGRADE TO A BLOCK IN STATE SHARED. WHEN IS THE RACE RESOLVED?
- A: AFTER BUS ARBITRATION

TRANSIENT STATES DEAL WITH RACE CONDITIONS THAT RESULT FROM E.G. RESOURCE ARBITRATION:



SPLIT-TRANSACTION BUS

- SPLIT-TRANSACTION BUS: PIPELINES A SEQUENCE OF PHASES IN A BUS TRANSACTION; E.G. ARBITRATION, TRANSFER, RESPONSE
- MUST DIVIDE TRANSACTION INTO SUBTRANSACTIONS: ADDS ADDITIONAL LATENCY; E.G. FOR BUS ARBITRATION:
- TRADEOFFS LATENCY VS BANDWIDTH
- MUST BALANCE PIPELINE STAGES TO MAXIMIZE THROUGHPUT
- SEPARATION BETWEEN REQUEST/RESPONSE ARBITRATION



ACCESS RACES ON SPLIT-TRANSACTION BUSES

MUST DEAL WITH

- MULTIPLE REQUESTS TO SAME ADDRESS
- REPORTING SNOOP RESULTS
- BUFFER OVERFLOWS

USE CONCEPT OF TRANSIENT STATES

MULTIPLE REQUESTS TO SAME ADDRESS:

- ALLOW ONLY ONE REQUEST TO SAME ADDRESS AT A TIME
- USE 'REQUEST TABLE' MONITORED BY ALL NODES

EARLY REPORT OF SNOOP RESULTS

- CACHE DIR. LOOKUP BEFORE BUFFERING IS ESSENTIAL

BUFFER OVERFLOW PREVENTION

- ADD PROVISIONS IN PROTOCOL TO 'NACK' WHEN BUFFERS OVERFLOW

MULTI-LEVEL CACHE ISSUES

ADDING ANOTHER LEVEL OF PRIVATE CACHE OFFERS SOME BENEFITS

- SHORTER MISS PENALTY TO NEXT LEVEL
- CAN FILTER OUT SNOOP ACTIONS TO FIRST LEVEL; ESPECIALLY IF INCLUSION IS MAINTAINED

INCLUSION IS NOT ALWAYS EASY TO MAINTAIN

- IF NOT MAINTAINED, IT CAN BE FORCED BY EVICTING A BLOCK AT LEVEL 1 WHEN BLOCK IS EVICTED AT LEVEL 2
- CAN CAUSE INCREASED PROCESSOR BLOCK OUT TO LEVEL 1

WRITE POLICY IS IMPORTANT TO REDUCE SNOOP OVERHEAD

- IF LEVEL 1 IS WRITE-BACK LEVEL 2'S COPY IS INCONSISTENT AND DIRTY MISS REQUESTS MUST BE SERVICED BY LEVEL 1
- IF LEVEL 1 IS WRITE-THROUGH AND INCLUSION IS MAINTAINED, LEVEL 2 CAN ALWAYS RESPOND TO MISS REQUESTS FROM OTHER PROCESSORS; CAN REDUCE OVERHEAD

TRUE vs FALSE SHARING

- ASSUME THAT A BLOCK IS SHARED BY TWO PROCESSORS
  - THE BLOCK CONTAINS TWO WORDS, word1 AND word2
- TRUE SHARING: THE TWO PROCESSORS ACCESS THE SAME WORD

FINE GRAIN SHARING:

P1	P2
W1	
	R1
	W1
W1	
	W1
	R1
	W1
R1	

UPDATE PROTOCOL IS BETTER

COARSE GRAIN SHARING

P1	P2
W1	
R1	
R1	
W1	
W1	
	W1
	R1
	R1

INVALIDATE PROTOCOL BETTER

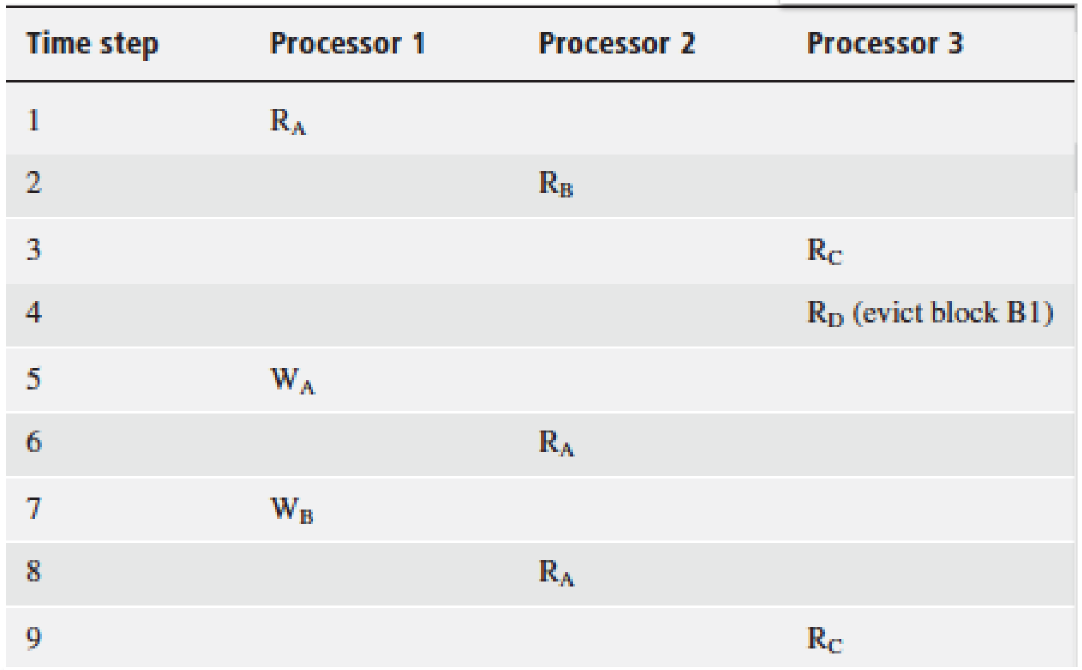
TRUE SHARING COMMUNICATES VALUES – ESSENTIAL

- FALSE SHARING: THE TWO PROCESSORS ACCESS DIFFERENT WORDS IN THE BLOCK
  - IF P1 ACCESSES W1 AND P2 ACCESSES W2. THEN SHARING IS USELESS
  - WRITE INVALIDATE CAUSES FALSE SHARING MISSES
  - WRITE UPDATE CAUSES FALSE SHARING UPDATES TO DEAD COPIES

FALSE SHARING DOES NOT COMMUNICATE VALUES
USELESS (NON-ESSENTIAL). PURE OVERHEAD

ESSENTIAL VS NON-ESSENTIAL MISSES

EXAMPLE) ASSUME A,B AND C BELONG TO SAME BLOCK (B1) AND D TO ANOTHER



Time step	Processor 1	Processor 2	Processor 3
1	R_A		
2		R_B	
3			R_C
4			R_D (evict block B1)
5	W_A		
6		R_A	
7	W_B		
8		R_A	
9			R_C

COLD MISS

COLD MISS

COLD MISS

COLD MISS

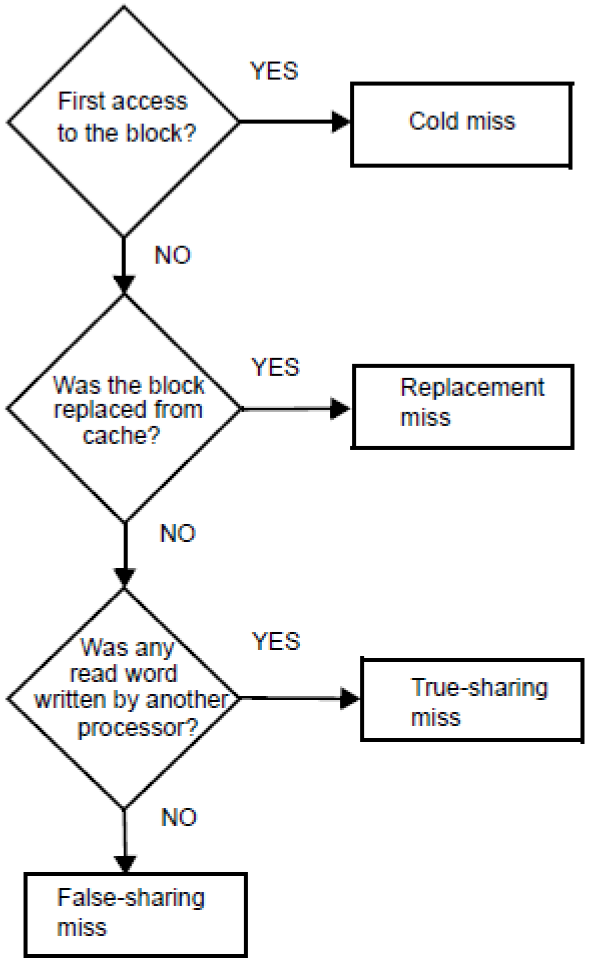
TRUE SHARING MISS

FALSE SHARING MISS

REPLACEMENT MISS

- COLD, TRUE AND REPLACEMENT (CONFLICT, CAPACITY) ARE ESSENTIAL WHEREAS FALSE SHARING MISSES ARE NON-ESSENTIAL; CAN BE IGNORED
- SAME REASONING CAN BE APPLIED TO MEMORY TRAFFIC

CLASSIFICATION OF MISSES


First access to the block?
YES
Cold miss
NO
Was the block replaced from cache?
YES
Replacement miss
NO
Was any read word written by another processor?
YES
True-sharing miss
NO
False-sharing miss

TRANSLATION LOOKASIDE BUFFER (TLB) CONSISTENCY

RECALL

- VIRTUAL MEMORY TYPICALLY MANAGES THE MAIN MEMORY "CACHE" BY MIGRATING FIXED-SIZED MEMORY PAGES BETWEEN MAIN MEMORY AND DISK
- PAGE TABLES (STORED IN MAIN MEMORY) KEEP TRACK OF THE MAIN-MEMORY RESIDENT PAGES
- A (VIRTUAL) MEMORY ADDRESS IS TRANSLATED TO A (PHYSICAL) MAIN MEMORY ADDRESS. NEEDS HARDWARE SUPPORT!
- A TLB IS A CACHE FOR VIRTUAL-TO-PHYSICAL TRANSLATIONS AND TYPICALLY SITS BETWEEN PROC. AND ITS LEVEL 1 CACHE

PAGE TABLE ENTRIES CAN CHANGE ON

- PAGE EVICTIONS
- ACCESS-RIGHT CHANGES
- ACCESS STATS CHANGES (REF BITS, DIRTY BITS)

IN A MULTIPROCESSOR, TLBs ACT AS PRIVATE CACHES

CONSISTENCY BETWEEN TLBS IS ESSENTIAL FOR CORRECTNESS

ENFORCING TLB CONSISTENCY

TLB IS INCONSISTENT WHEN ONE OF ITS ENTRIES IS NOT CONSISTENT WITH THE MASTER COPY – THE PAGE TABLE ENTRY

SOURCES OF INCONSISTENCY

- A PAGE IS EVICTED
- ACCESS RIGHTS ARE CHANGED
- ACCESS STATISTICS IS CHANGED (E.G. PAGE BECOMES DIRTY)

NOT ALL ARE CRITICAL FOR CORRECTNESS

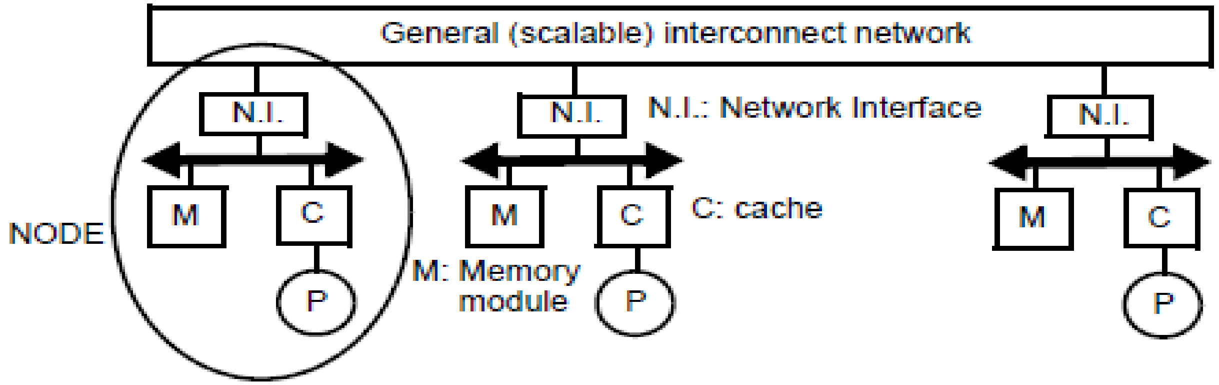
ONE SOLUTION: TLB SHOOTDOWN

1. ON PAGE EVICTION, MASTER PROC. LOCKS PAGE TABLE ENTRY
2. MASTER PROC. INTERRUPTS ALL INVOLVED PROCESSORS
3. EACH PROC. INVALIDATES THE INCONSISTENT TLB ENTRY
4. EACH PROC. INVALIDATES INCONSISTENT CACHE ENTRIES
5. EACH PROC. ACKS BACK TO MASTER PROC.

TLB SHOOTDOWN IS AN EXPENSIVE OPERATION; LUCKILY IT HAPPENS RARELY

SCALABLE CACHE COHERENCE SOLUTIONS

BUS-BASED MULTIPROCESSORS ARE INHERENTLY NON-SCALABLE



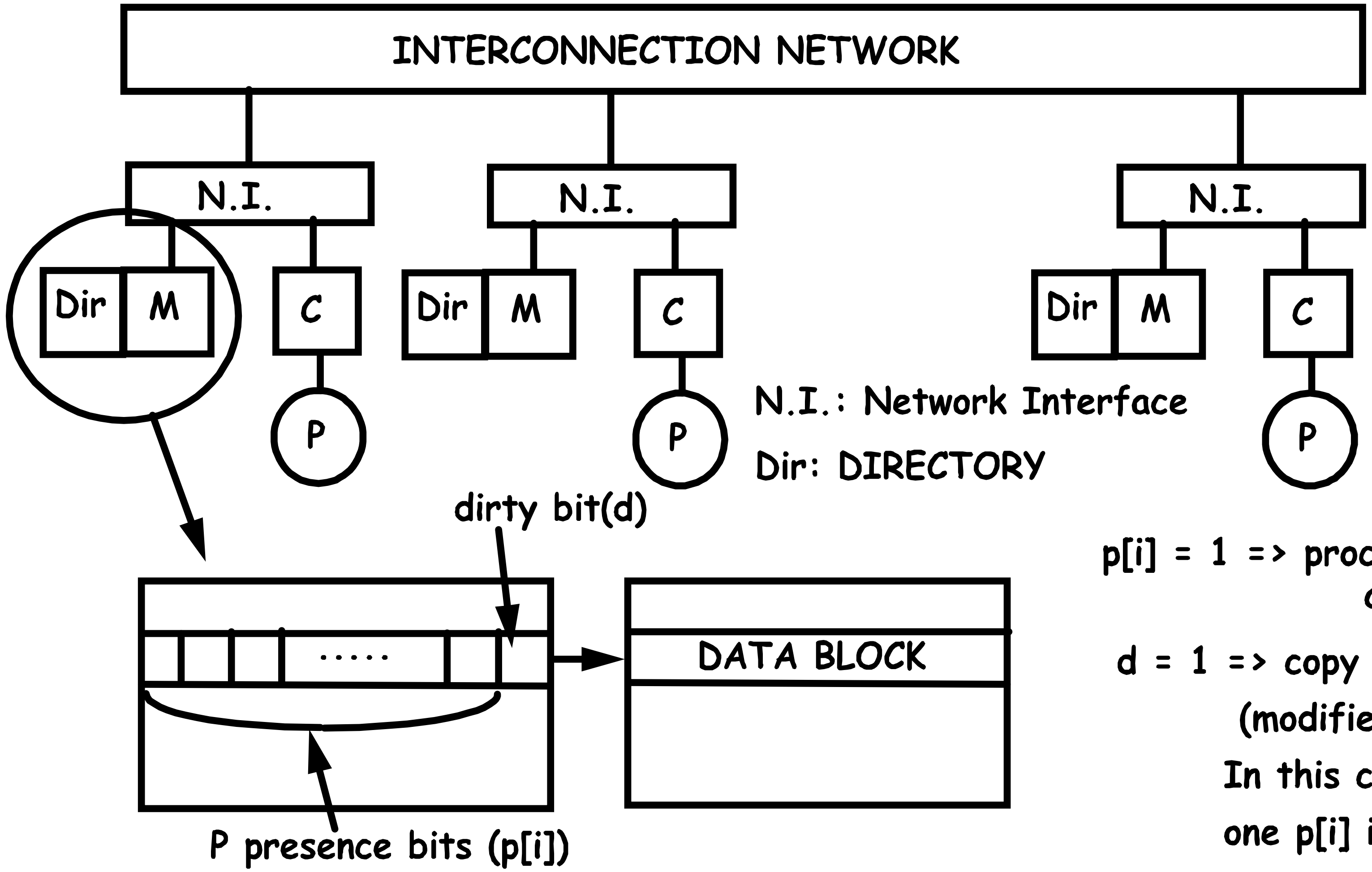
CACHE-COHERENT, NON-UNIFORM MEM. ARCHITECTURES (cc-NUMA):

- DISTRIBUTE MAIN MEMORY ACROSS NODES TO LEVERAGE LOCALITY
- USE A GENERAL (SCALABLE) INTERCONNECT TO ACCOMMODATE BANDWITH

SNOOPY (BROADCAST) BASED CACHE PROTOCOLS ARE NOT SCALABLE

SCALABLE CACHE PROTOCOLS SHOULD KEEP TRACK OF SHARERS

DIRECTORY-BASED PROTOCOLS



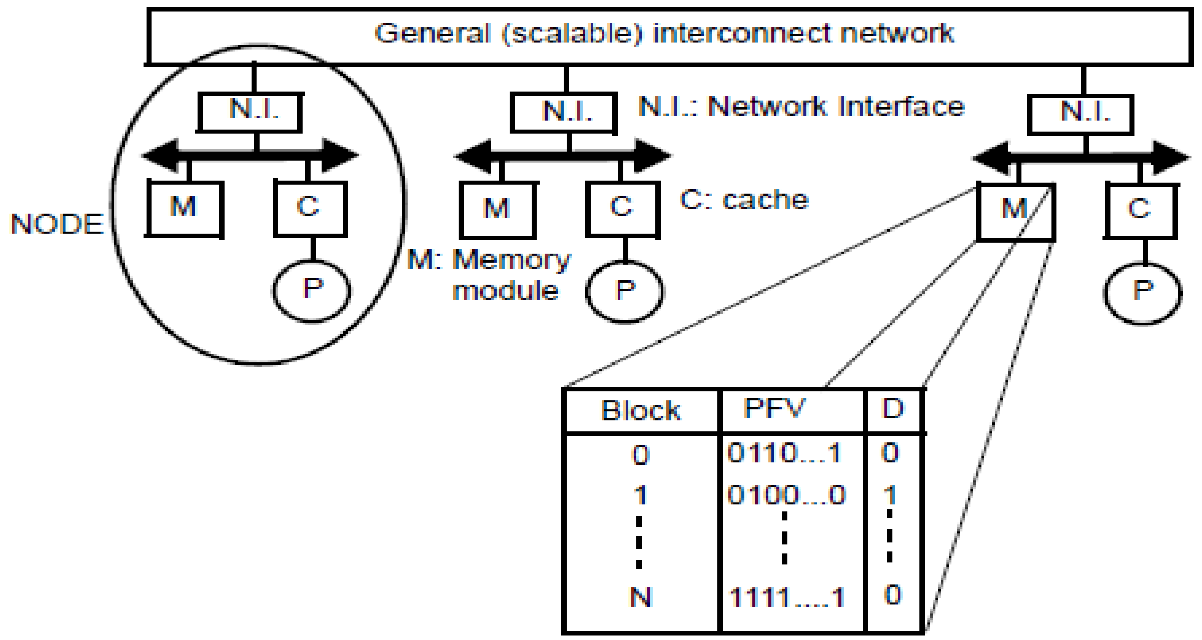
N.I.: Network Interface

Dir: DIRECTORY

p[i] = 1 => proc i has copy

d = 1 => copy is dirty (modified). In this case only one p[i] is set

PRESENCE-FLAG VECTOR SCHEME



Block	PFV	D
0	0110...1	0
1	0100...0	1
⋮	⋮	⋮
N	1111....1	0

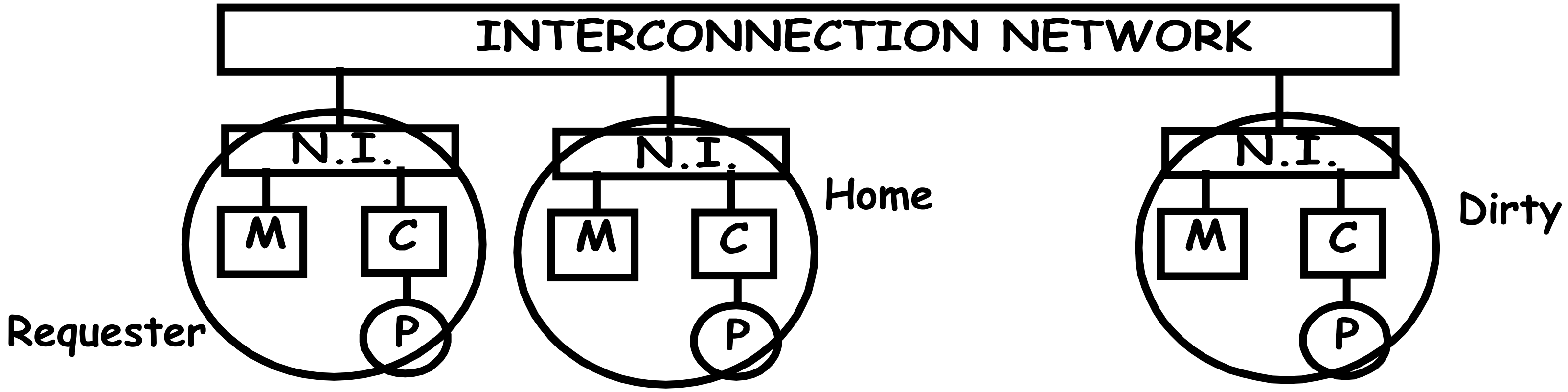
EXAMPLE)

- BLOCK 1 IS CACHED BY PROC. 2 ONLY AND IS DIRTY
- BLOCK N IS CACHED BY ALL PROCESSORS AND IS CLEAN

LET'S CONSIDER HOW THE PROTOCOL WORKS

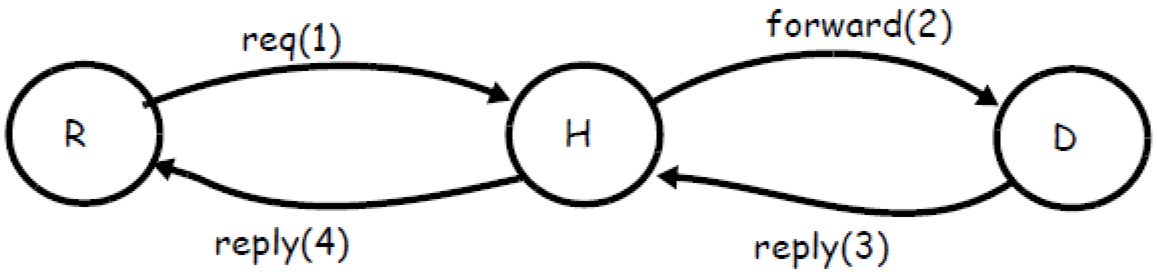
cc-NUMA PROTOCOLS

- **USE SAME PROTOCOL AS FOR SNOOPING PROTOCOLS (E.G. MSI-INVALIDATE, MSI-UPDATE OR MESI**
- **PROTOCOL AGENTS:**
  - HOME NODE (H): NODE WHERE THE MEMORY BLOCK AND ITS DIRECTORY ENTRY RESIDE
  - REQUESTER NODE (R): NODE MAKING THE REQUEST
  - DIRTY NODE (D): NODE HOLDING THE LATEST, MODIFIED COPY
  - SHARED NODE(S) (S): NODES HOLDING A SHARED COPY

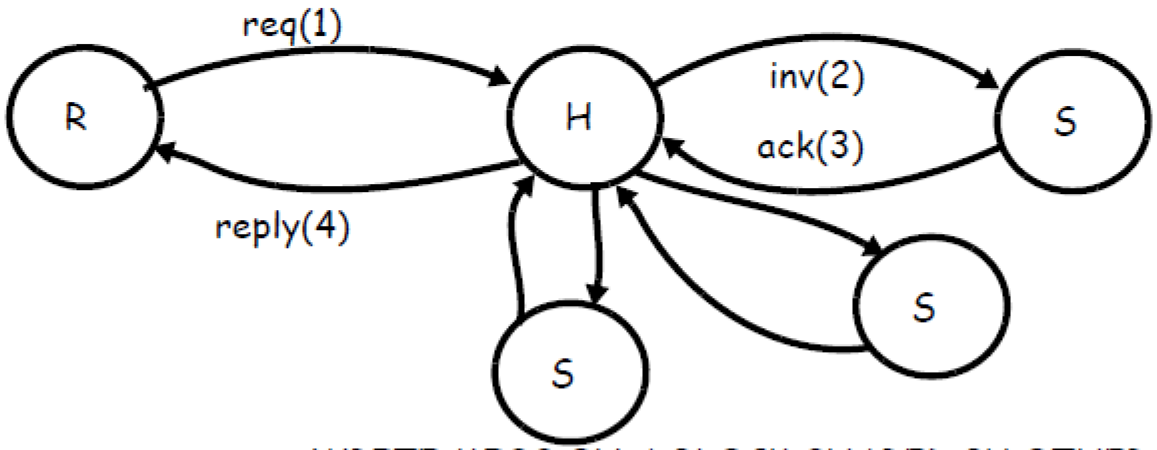


HOME MAY BE THE SAME NODE AS REQUESTER OR DIRTY
NOTE:BUSY BIT PER ENTRY

MSI INVALIDATE PROTOCOL IN cc-NUMAs



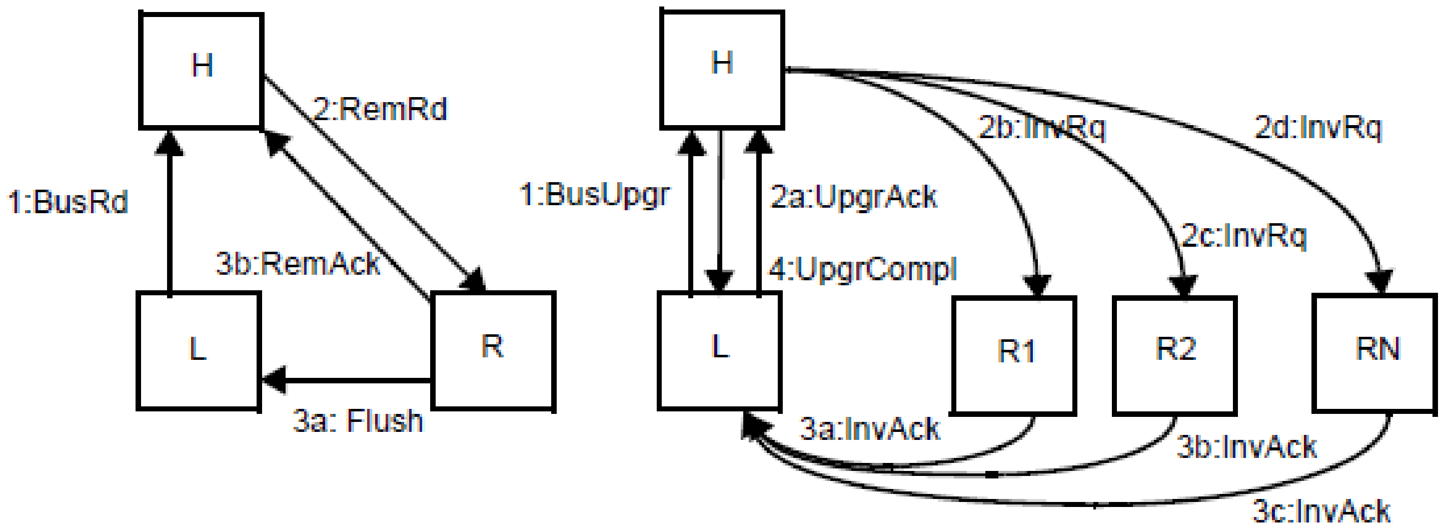
READ MISS ON A BLOCK DIRTY REMOTE



WRITE MISS ON A BLOCK SHARED BY OTHERS

NOTE: IN MSI-UPDATE THE TRANSITIONS ARE THE SAME EXCEPT THAT UPDATES ARE SENT INSTEAD OF INVALIDATIONS

REDUCING LATENCIES IN DIRECTORY PROTOCOLS



(a) BusRd transaction (b) BusUpgr transaction

BASELINE DIR. PROTOCOL INVOKES FOUR HOPS (IN WORST CASE) ON A REMOTE MISS. OPTIMIZATIONS ARE POSSIBLE

1. REQUEST IS SENT TO HOME
2. HOME REDIRECTS THE REQUEST TO REMOTE
3. REMOTE RESPONDS TO LOCAL
4. LOCAL NOTIFIES HOME (OFF THE CRITICAL ACCESS PATH)

MEMORY REQUIREMENTS OF DIRECTORY PROTOCOLS

MEMORY REQUIREMENT OF A PRESENCE-FLAG VECTOR PROTOCOL

- N PROCESSORS (NODES)
- M MEMORY BLOCKS PER NODE

SIZE OF DIRECTORY: $M \times N^2$

DIRECTORY SCALES WITH THE SQUARE OF NUMBER OF PROCESSORS; A SCALABILITY CONCERN!

ALTERNATIVES

- LIMITED POINTER PROTOCOLS: MAINTAIN *i* POINTERS (EACH log N BITS) INSTEAD OF N AS IN PRESENCE FLAG VECTORS

MEM. OVERHEAD: SIZE (DIR)/(SIZE(MEMORY) + SIZE(DIR))

EXAMPLE) MEM. OVERHEAD FOR LIMITED POINTER SCHEME:

$M \times N \times i \log N / (M \times N \times B + M \times N \times i \log N) =$

$i \log N / (B + i \log N)$

OTHER SCALABLE PROTOCOLS

COARSE VECTOR SCHEME

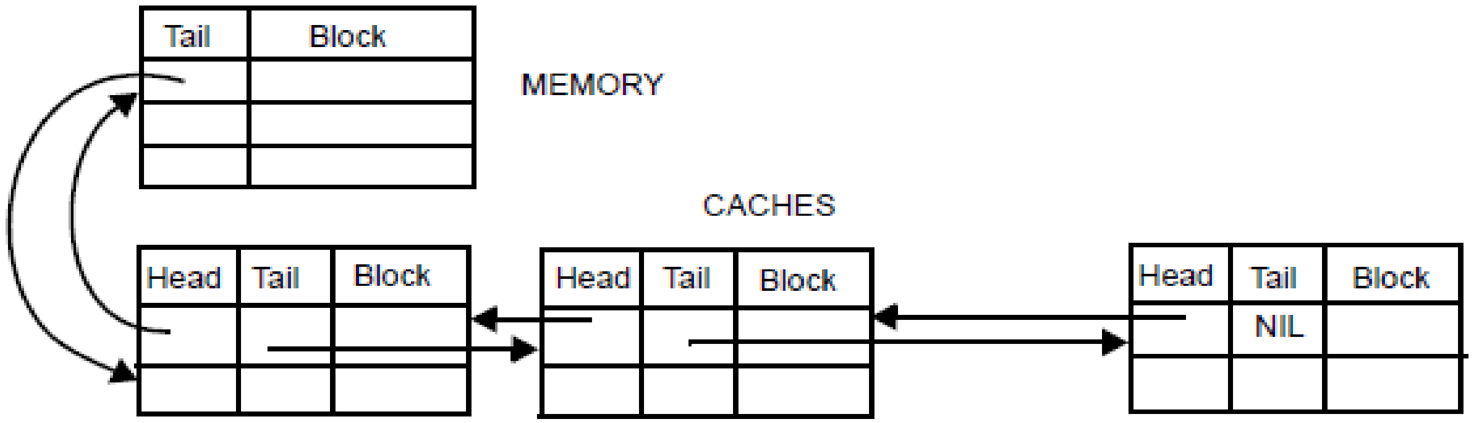
- PRESENCE FLAGS IDENTIFY GROUPS RATHER THAN INDIVIDUAL NODES

DIRECTORY CACHE

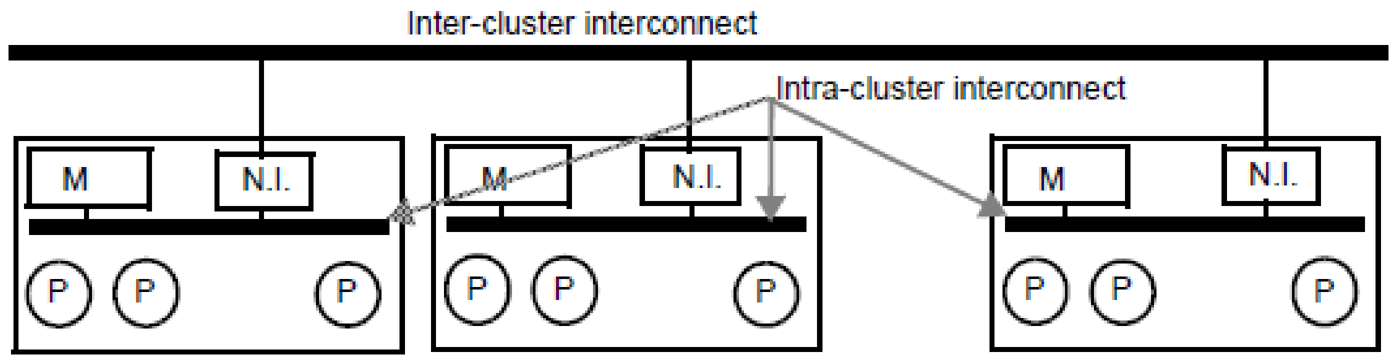
- DIRECTORY OVERHEAD IS PROPORTIONAL TO DIR. CACHE SIZE INSTEAD OF MAIN MEMORY SIZE (LEVERAGING LOCALITY)

CACHE-CENTRIC SCHEMES. MAKE OVERHEAD PROPORTIONAL TO (PRIVATE) CACHE SIZE INSTEAD OF MEMORY

EXAMPLE) SCALABLE COHERENT INTERFACE



HIERARCHICAL SYSTEMS



INSTEAD OF SCALING IN A FLAT CONFIG. WE CAN FORM CLUSTERS IN A HIERARCHICAL ORGANIZATION

RELEVANT INSIDE AS WELL AS ACROSS CHIP-MULTIPROCESSORS

COHERENCE OPTIONS:

- INTRA-CLUSTER COHERENCE: SNOOPY/DIRECTORY
- INTER-CLUSTER COHERENCE: SNOOPY/DIRECTORY

TRADEOFFS AFFECT MEMORY OVERHEAD AND PERFORMANCE TO MAINTAIN COHERENCE