LECTURE 4

MULTIPROCESSOR SYSTEMS (I)

Miquel Pericàs

EDA284/DIT361 - 2019/2020

This week

1. **Lectures**
   - Shared Memory Multiprocessors (L4 today + L5 tomorrow 8h)
   - Roofline Model (L6 Friday 8h)
2. **Practice session #1 (Friday 10h)**
   - SMPs & Performance Metrics
   - Exercises will be published tomorrow
3. **Project**
   - Today is last day to propose a team and select three scenarios. Tomorrow we will setup the final teams.
   - If you have not submitted your preference, you will be assigned a random team member and scenario
4. **Small change in schedule**
   - Practice session #2 has been moved from Feb 19th (Wed) to Feb 18th (Tue). Currently there is no lecture scheduled on Feb 19th

OUTLINE (Lectures 4+5)

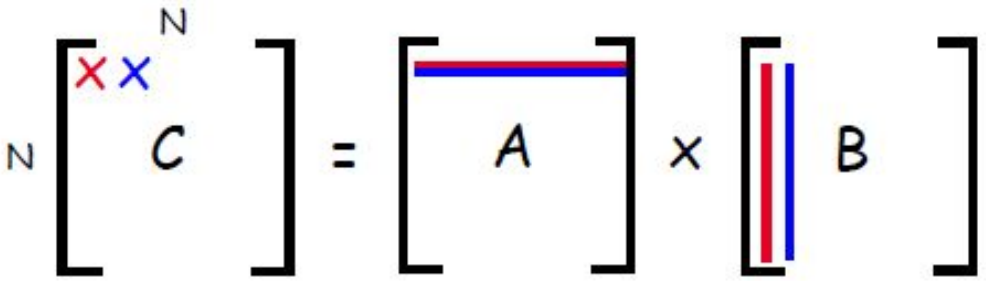
- Shared-memory parallel programming
- Bus-based shared memory systems
  - architectures
  - coherence protocols
  - coherence optimizations
  - multi-level caches
- Classification of Misses
- Scalable shared memory systems

EXAMPLE: MATRIX MULTIPLY + SUM

sequential program:

```
1                sum = 0;
2                for (i=0;i<N;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                            sum += C[i,j];
8                            }
```

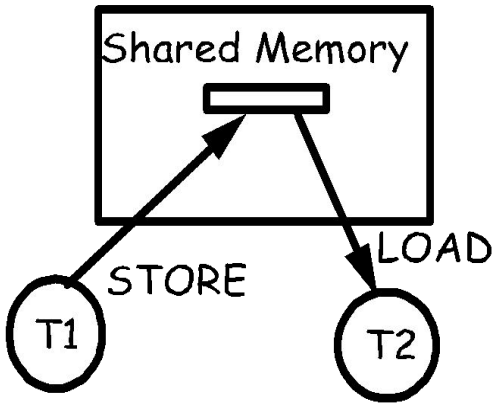
- multiply matrices A[N,N] by B[N,N] and store result in C[N,N]
- add all elements of C[,] and store in variable 'sum'



How to parallelize?

TWO TYPES OF INTER-PROCESSOR COMMUNICATION

- **Implicitly 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



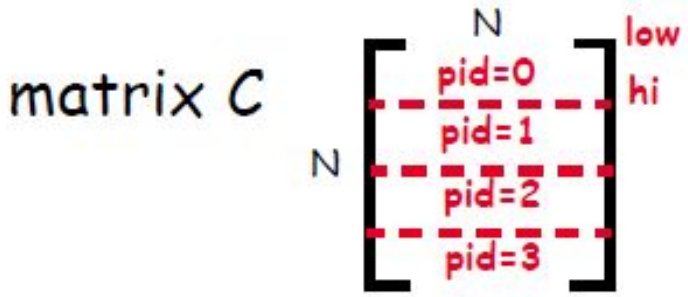
- **Explicitly via messages (sends and receives)**
  - need to know the destination and what to send
  - explicit message-passing statements in the code
  - called “message passing”

NO HYPOTHESIS ON THE RELATIVE SPEED OF PROCESSORS

SHARED MEMORY EXAMPLE: MATRIX MULTIPLY + SUM

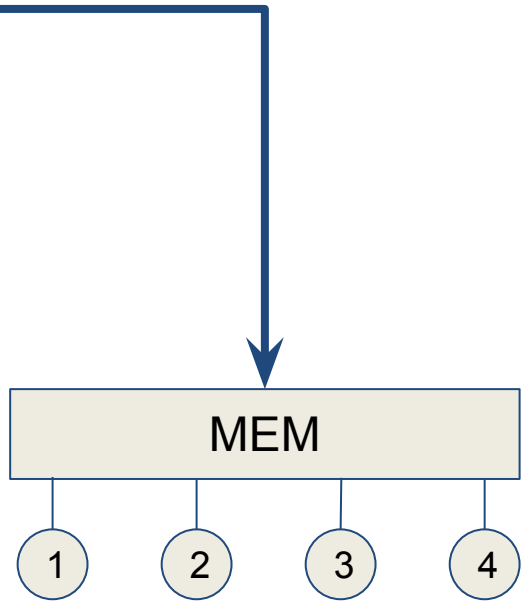
SHARED-MEMORY PROGRAM

```
      /* A, B, C, BAR, LV and sum are shared
      /* All other variables are private
1a    low = get_thread_num()*N/nproc; //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);
```



what extensions to the single-threaded program are needed to execute this parallel version?

```
/* A, B, C, BAR, LV and sum are shared
/* All other variables are private
1a   low = get_thread_num()*N/nproc; //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);
```



what extensions to single-threaded CPU are needed to execute this program?

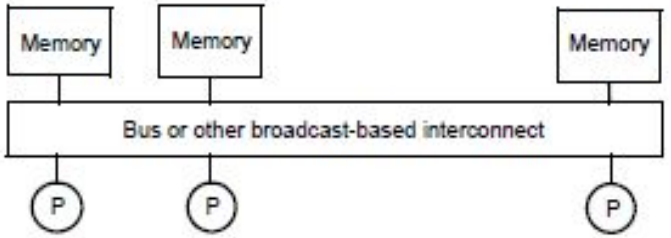
1. Need to correctly communicate data, in timely fashion. Stores need to be propagated to Loads:
   - in the same thread, and
   - across different threads

Today's lecture

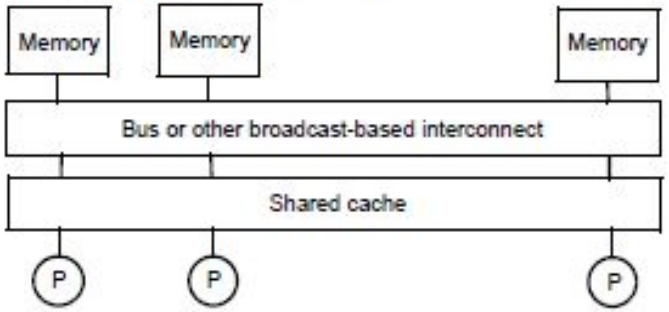
2. Implement interconnect + caches for performance:
   - private caches for fast access
   - shared caches for global data (multithreading) and dynamic sharing of cache space (multiprogrammed)
3. Need to provide constructs for synchronization (barriers and locks)
   - Lecture 13
4. Need a method to create threads and to obtain thread numbers (this is the domain of O/S)
   - discussed partially in DAT400

BUS-BASED SHARED MEMORY SYSTEMS

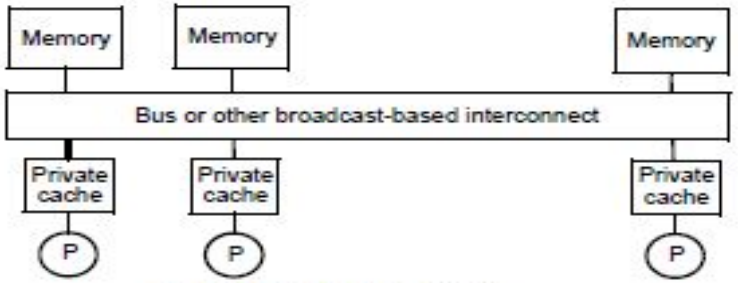
Bus: broadcast-based interconnect medium



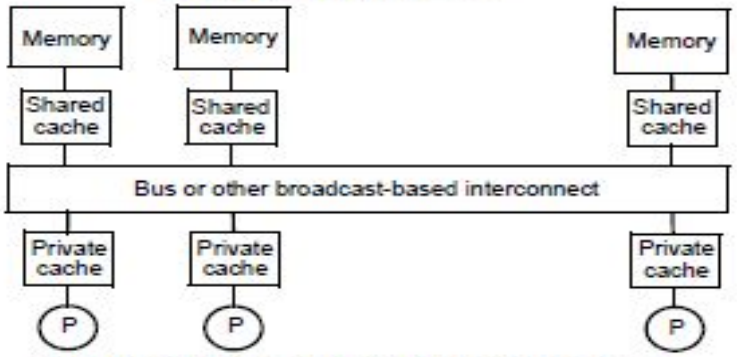
(a) Dance-hall multiprocessor architecture or SMP



(b) SMP with shared level 1 cache



(c) SMP with private caches



(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 private (c)

(c) private L1 caches; lower latency but smaller cache capacity

(d) private L1 caches and shared L2; popular for chip multiprocessors (CMPs)

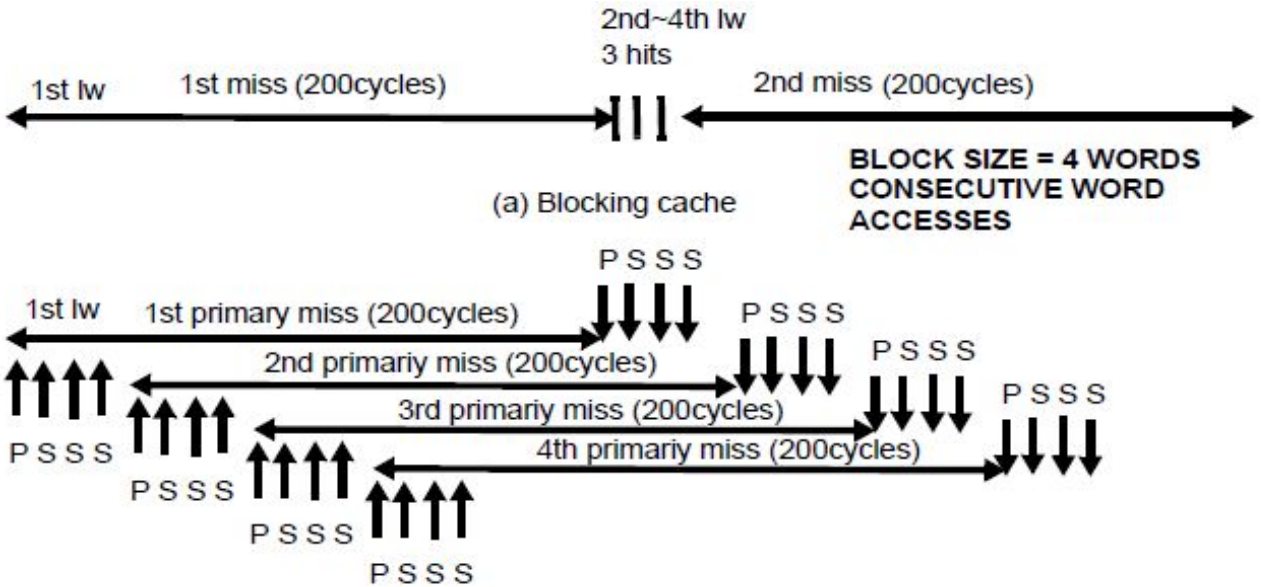
rest of discussion considers organization (c) and (d)

RECAP ON MULTI-LEVEL CACHES

- **multi-level cache allocation policies**
  - inclusive vs exclusive vs non-inclusive
- **write policies**
  - next slide
- **non-blocking**
  - continue servicing loads in the shadow of a miss

BLOCKING CACHE

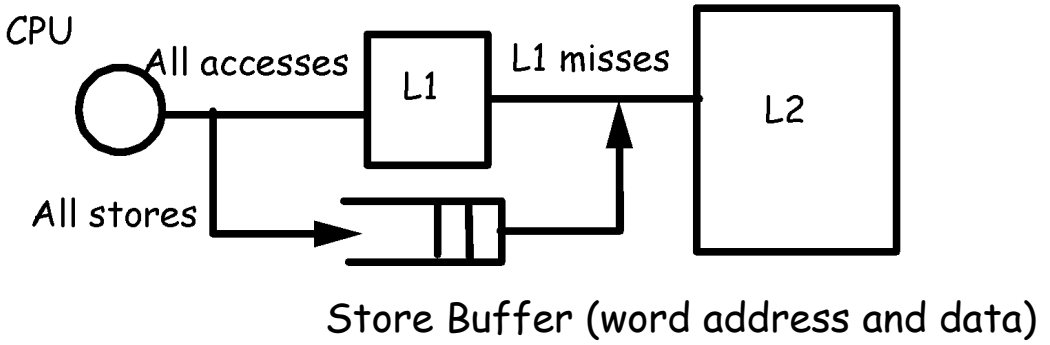
NON-BLOCKING CACHE



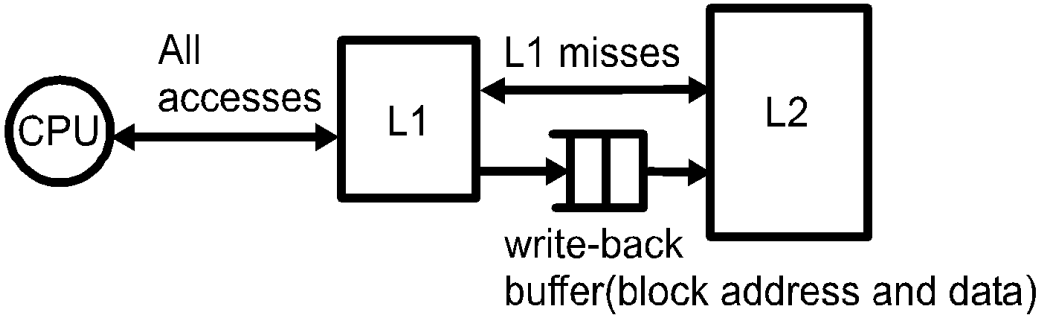
(b) Non-blocking cache with 16 MSHRs

HOW TO HANDLE WRITES?

- **WRITE THROUGH:** write to next level on all writes
  - combined with store buffer (SB) to avoid CPU stalls. SB holds individual STs
  - simple, no inconsistency among levels.



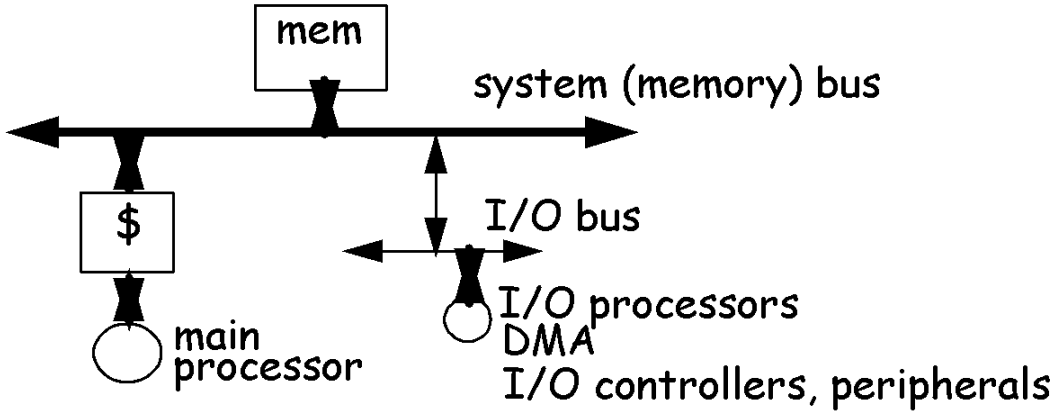
- **WRITE-BACK:** write to next level on replacement (reduces write traffic)
  - with the help of a dirty bit and a write-back buffer. WB holds dirty cache lines
  - writes happen in background on a miss only



- **ALLOCATION ON WRITE MISSES**
  - always allocate in write-back; design choice in write-through
  - small caches often use WT + no-alloc to alleviate storage pressure

COHERENCE

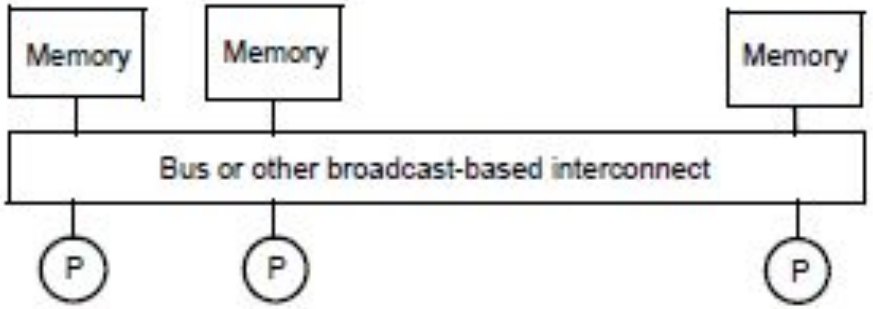
- **in uniprocessors, a load must return the value of the latest store in process order with the same address**
  - this is done through memory disambiguation (**3.4.5**) and management of cache hierarchy.
- **coherence between I/O traffic and cache must be enforced**
  - however, this is infrequent and software is informed
  - some solutions: uncacheable memory, uncacheable ops, cache flushing
  - also possible to pass I/O through cache



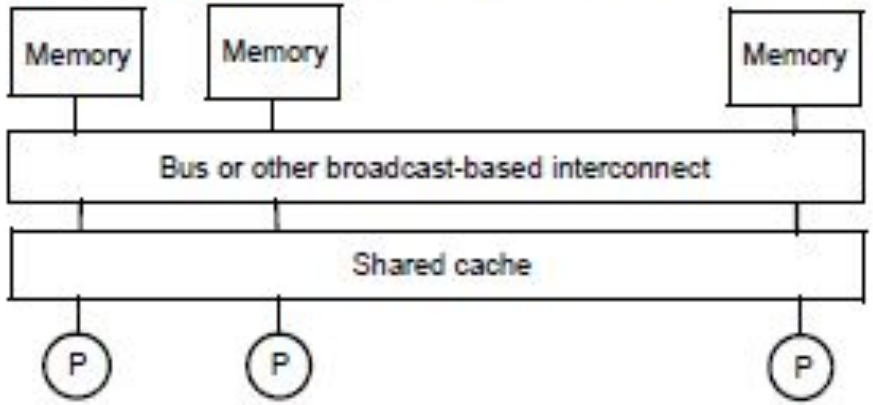
COHERENCE

- **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
  - coherence will be formally treated in lecture #14
- **informal definition (for multiprocessors)**
  - a cache system is coherent if all processors, at any point in time, have a consistent view of the last globally written value to each location
    - i.e. no processor can read an old value w_1 after w_2 has been globally performed
    - performed := value at memory has been altered

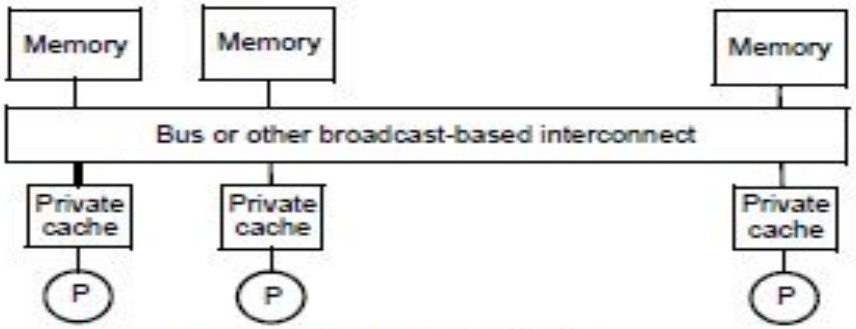
CACHE COHERENCE



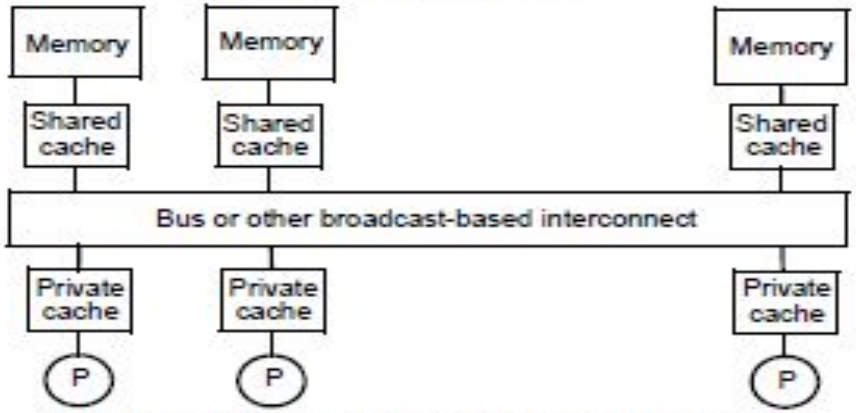
(a) Dance-hall multiprocessor architecture or SMP



(b) SMP with shared level 1 cache



(c) SMP with private caches

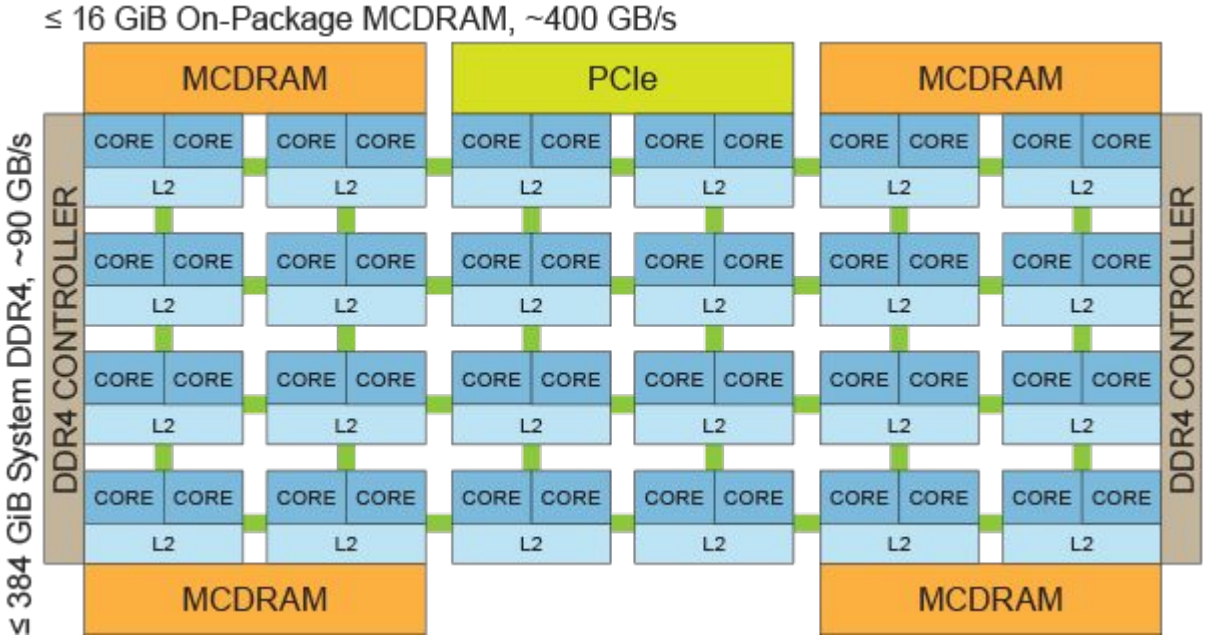


(d) SMP with private caches and shared Level 2 cache

which of these systems are more prone to suffer coherence problems?

those that can contain multiple copies of the same location: (c) and (d)

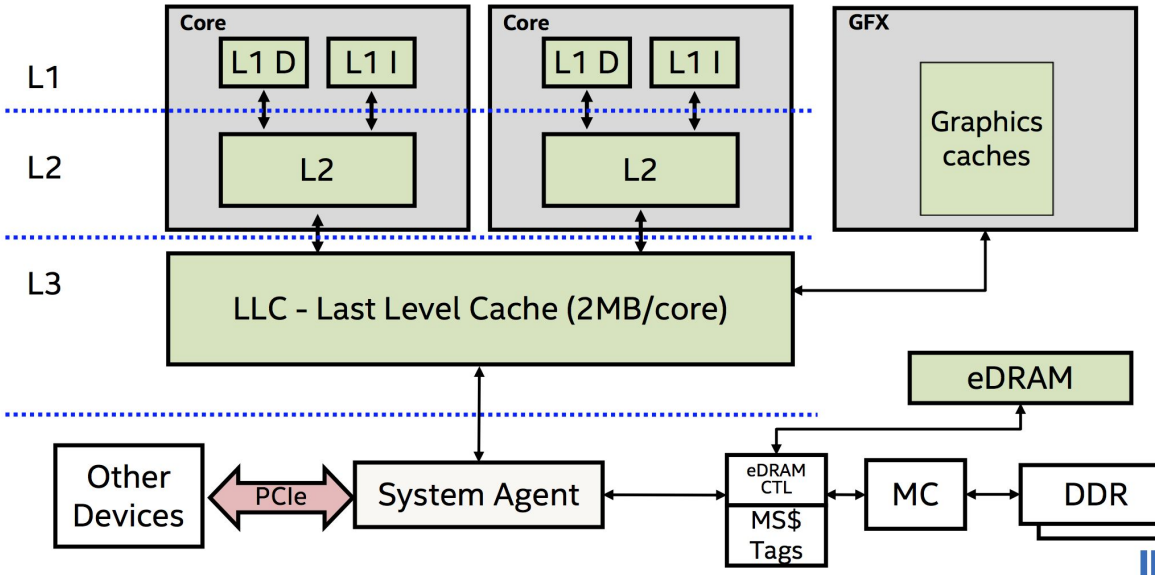
IS CACHE COHERENCE A PROBLEM IN MODERN PROCESSORS?



Intel Knights Landing
(SMP with private caches)

YES: multiple cache levels, replicated L1&L2 caches

eDRAM Based Cache

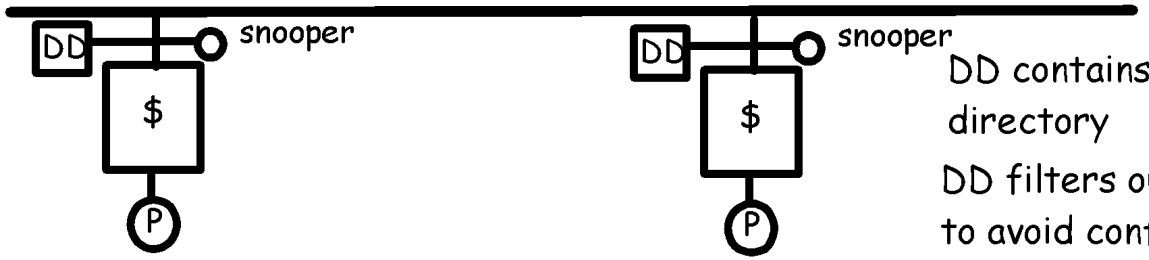


Intel Skylake

(SMP with private L1/L2 caches and shared L3 cache)

SNOOPING-BASED CACHE COHERENCE

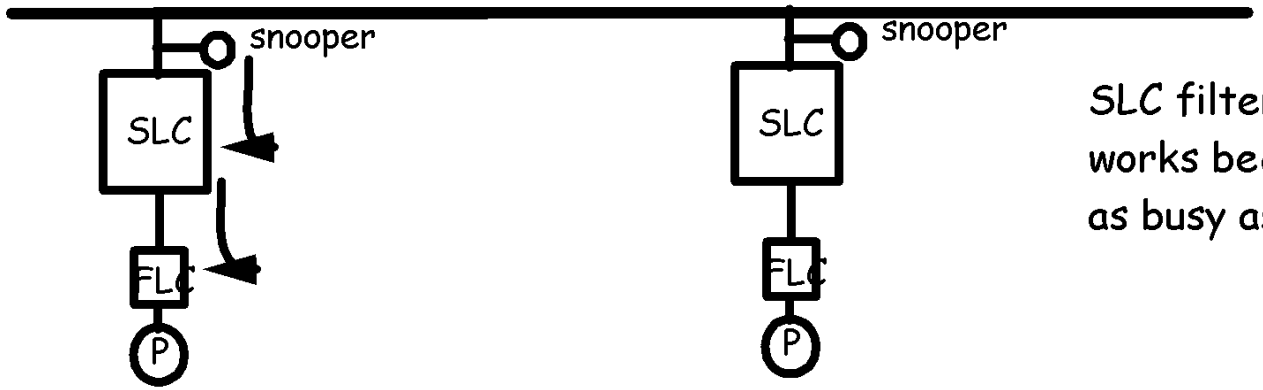
- **Basic idea**
  - Transactions on the bus are “visible” to all processors
  - Bus interface can “snoop” (monitor) the bus traffic and take action when required
  - To take action the “snooper” must check the status of the cache: competes with regular loads
- **Reduce cache controller congestion with a 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 (SLC)**
  - when there is inclusion, SLC filters out transactions to FLC (first level cache)
  - SLC contains a bit indicating whether the block is in FLC



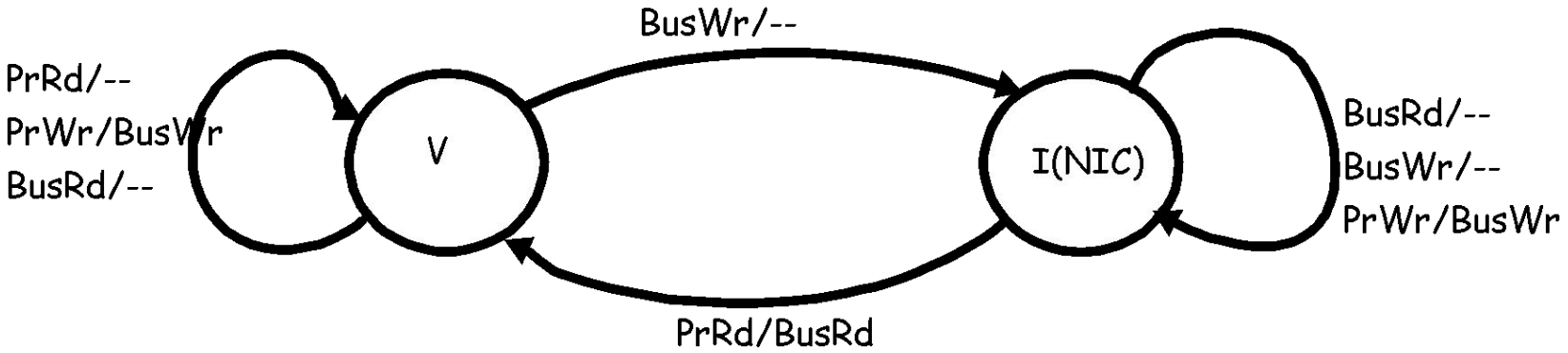
SLC filters out bus request works because SLC is not as busy as FLC

(in modern systems: SLC often called LLC := last level cache)

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**

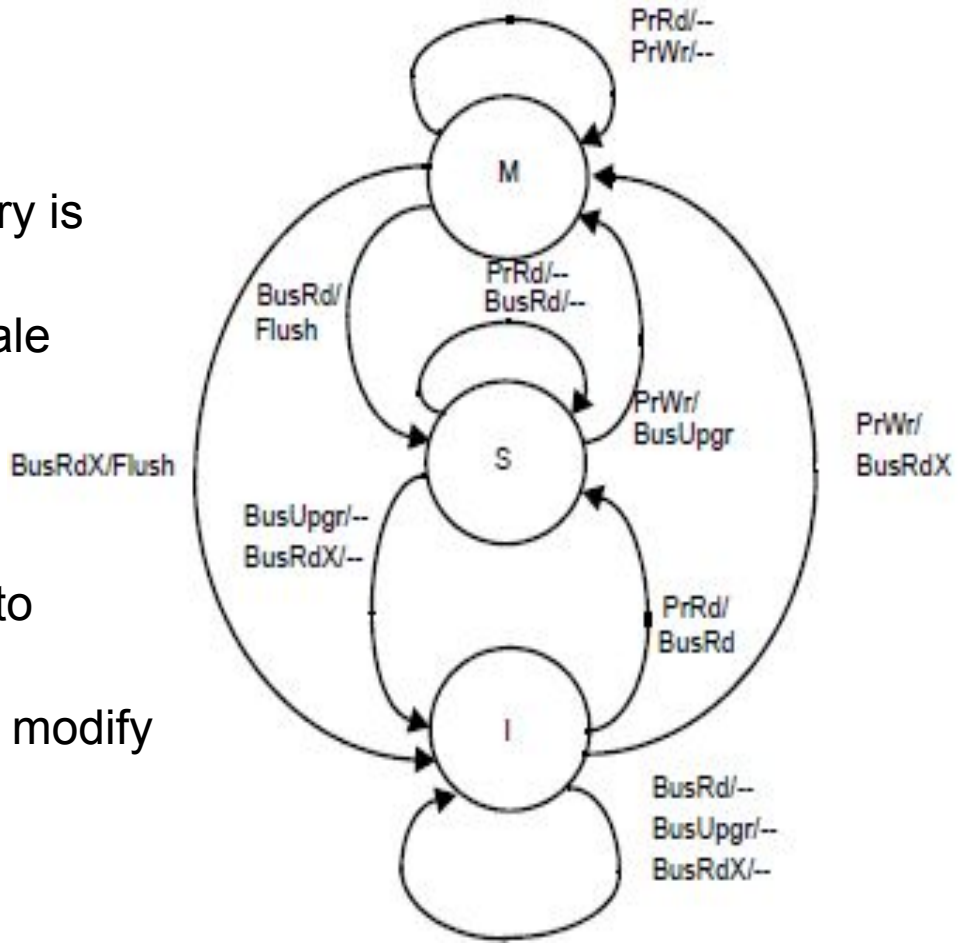
- **state diagram**
  - each cache is represented by a finite state machine
  - imagine **P** identical FSMs working together, one per cache
  - FSM represents the behavior of a cache w.r.t. a memory block
    - not the cache controller!



- **PrRd**/**PrWr**: Processor Read/Write; **BusRd**/**BusWr**: Bus Read/Write
- Good: (almost) no new state (1 bit per block in cache)
- Bad: most blocks are not shared, all write requests launch bus transactions → **generates unnecessary write traffic!**

WRITE-BACK: MSI INVALIDATE PROTOCOL

- **BLOCK STATES:**
  - Invalid (I);
  - Shared (S): one copy or more, memory is clean;
  - 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



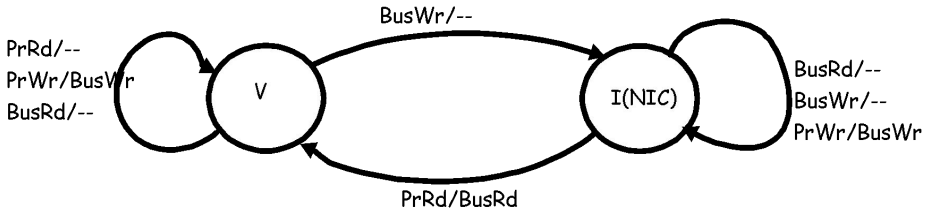
- write to shared block: use BusUpgr
- flush: forward block copy to requester (memory copy is stale)
  - memory should be updated at the same time
- Writes to blocks in state M do not generate Bus activity!

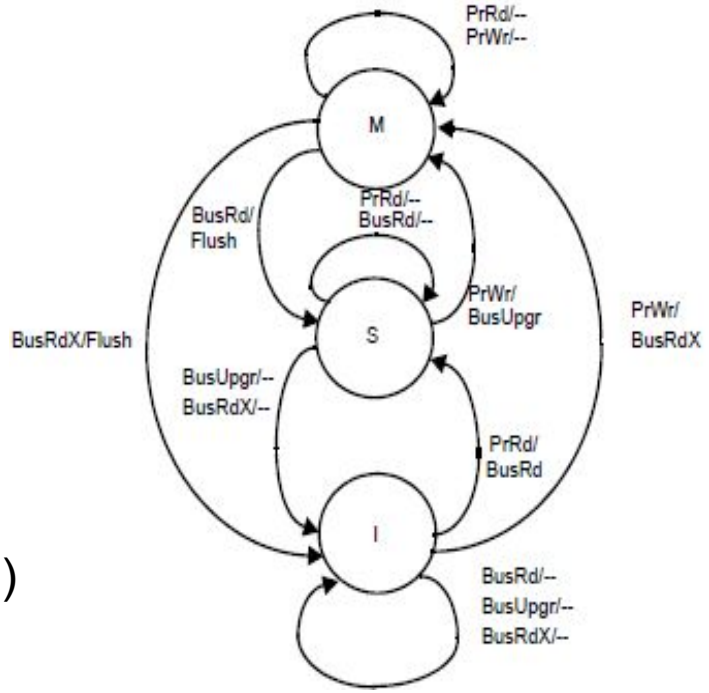
Exercise

Two processors P1 and P2 perform following series of Loads and Stores to a memory block storing word A

P1: LDA STA
P2: LDA
time

What are the coherence transitions and bus actions as performed by the cache of P1 for the two protocols **V**alid-**I**nvalid and **MSI**?





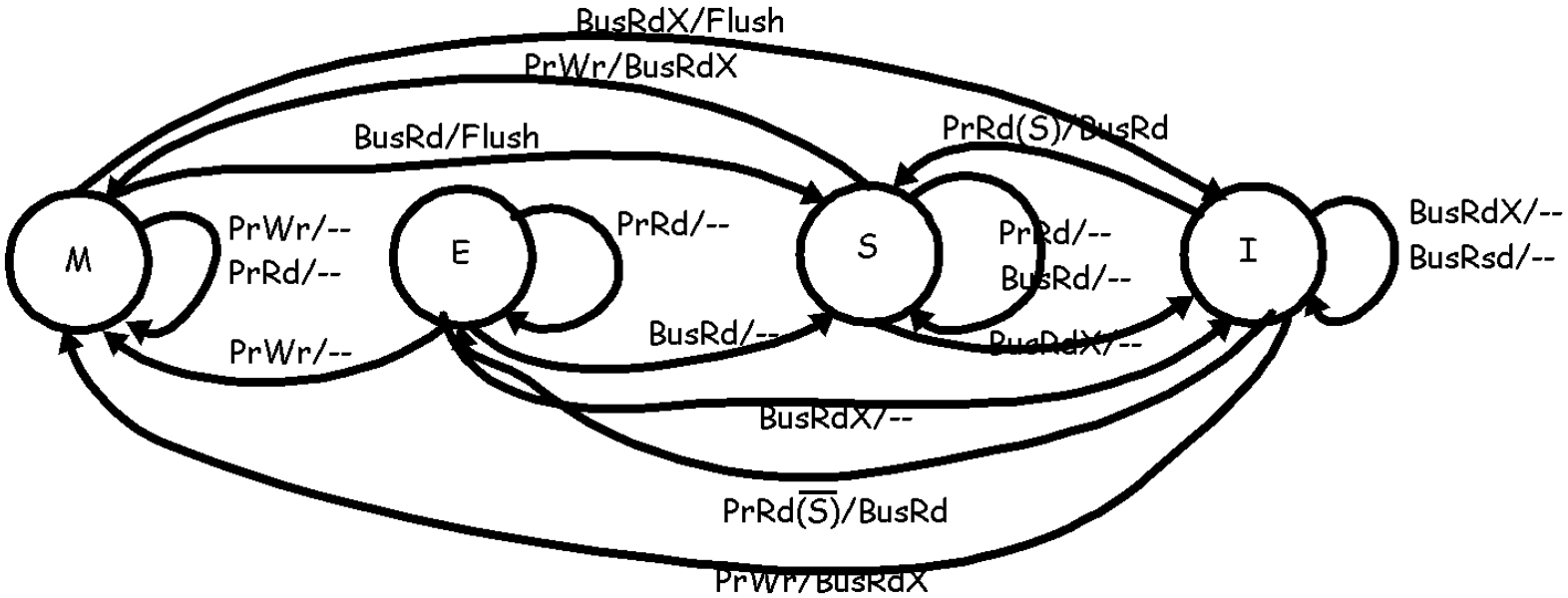
Transitions are as follows:

VI (P1): I->V (BusRd), V->V (BusWr), --

MSI (P1): I->S (BusRd), S->M (BusUpgr), M->S (Flush)

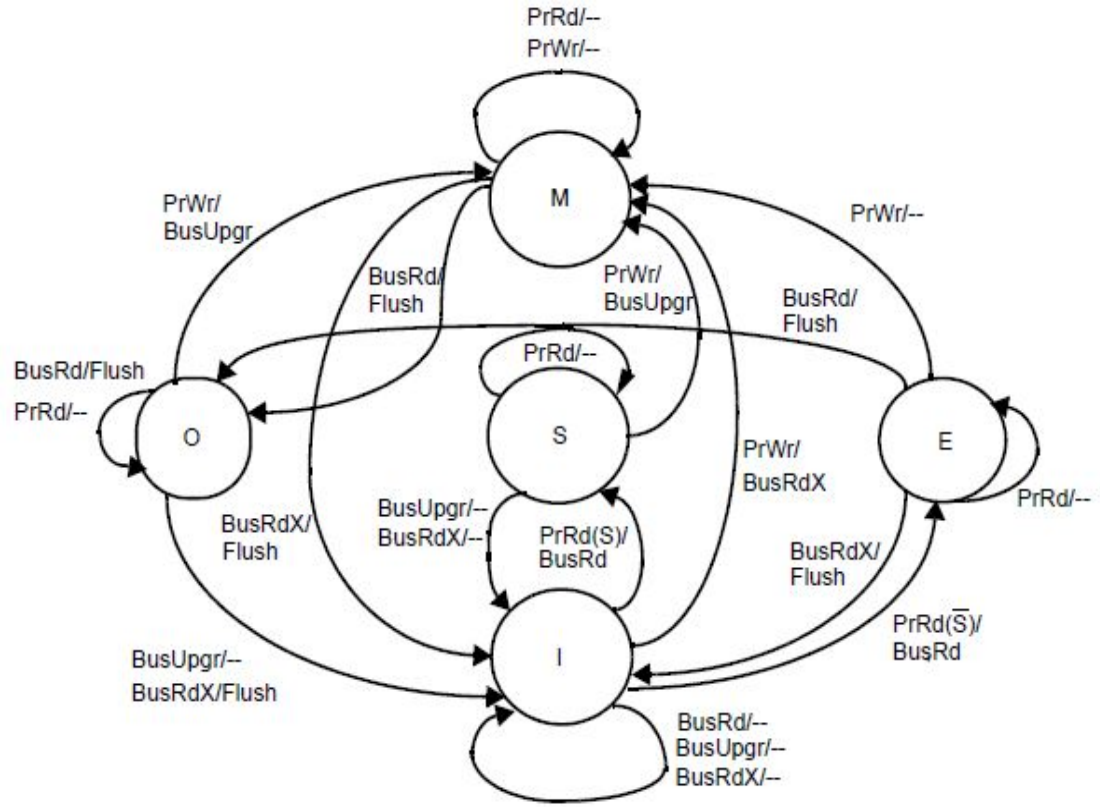
MESI PROTOCOL

- **problem with MSI invalidate protocol**
  - most of blocks are private--not shared (ex. multiprogrammed workloads)
  - read miss (LD @A) followed by write (ST @A) causes two bus accesses (BusRd, BusWr)
- **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.
- Transition from E to M is silent: No extra Bus Access
  - Could use BusUpgr in local transition from S to M

MOESI: A GENERAL CLASS OF PROTOCOLS



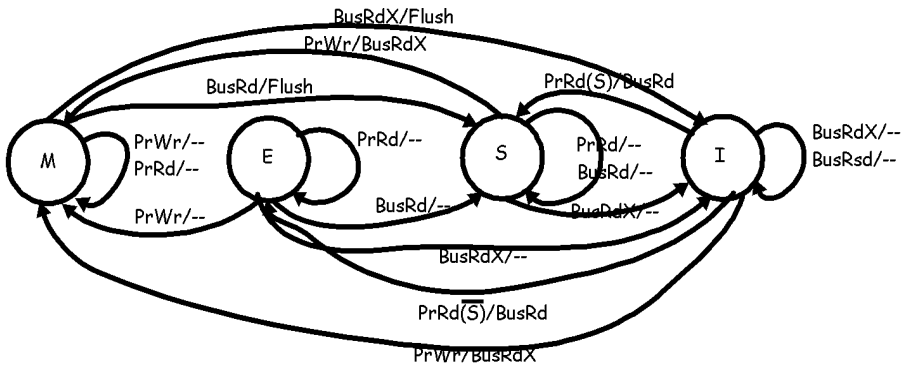
- MOESI can model (among others) MSI and MESI
- adds notion of **ownership** of a cache block
  - E&M implicitly owner. O used for ownership of 'S' blocks
- owner supplies a copy of the block when another cache requests it (enables sharing of dirty data, no Mem WB needed)
- ownership is transferred to another cache or to memory when block is invalidated/replaced (triggers WB)

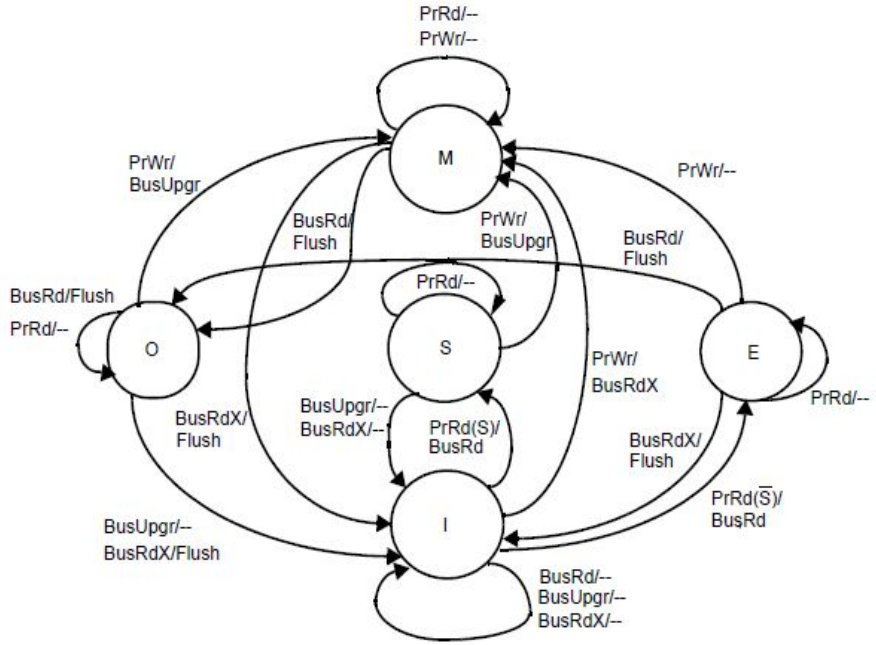
Exercise

Two processors P1 and P2 perform following series of Loads and Stores to a memory block storing address @A

P1: LDA STA
P2: LDA

What are the coherence transitions and bus actions as performed by the cache of P1 for the two protocols MESI and MOESI?





Transitions are as follows:
MESI: I->E (BusRd), E->M, M->S (Flush)
MOESI: I->E (BusRd), E->M, M->O (Flush)

CACHE COHERENCE PROTOCOLS IN USE

Some vendors have provided indication of the protocols in use

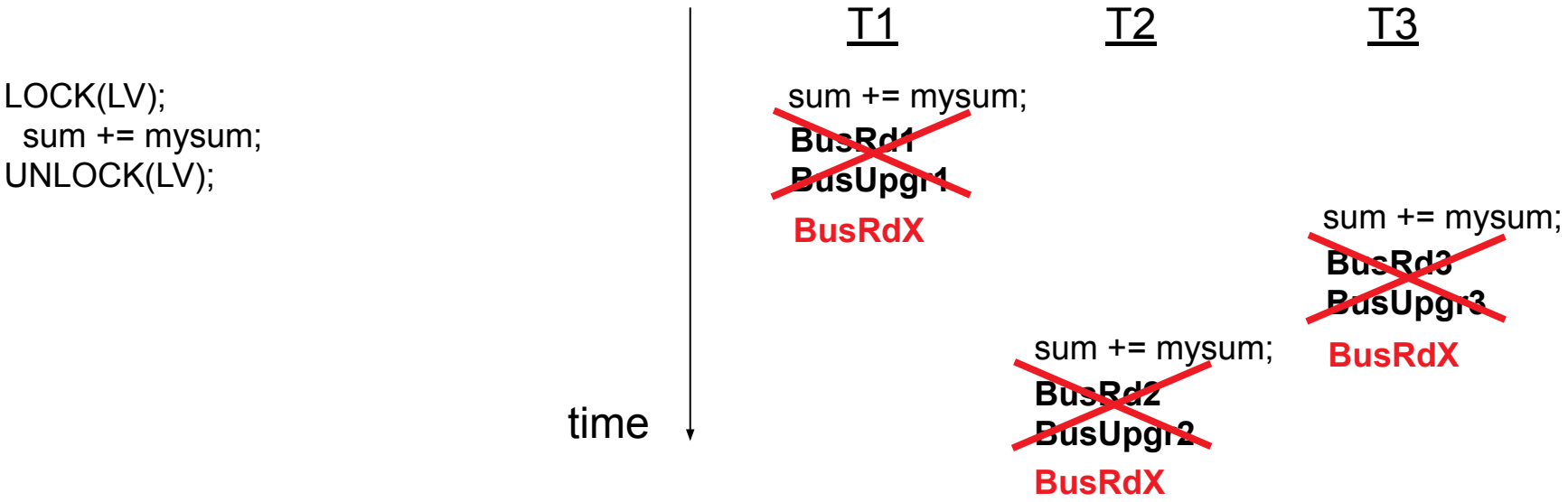
- **AMD64**, **ARMv8** (except Cortex-A9) use **MOESI**
- **intel64** uses **MESIF** (F = forwarding: similar to O but ownership transferred to requester along with data reply. exploits temporal locality)
- **Intel itanium 2, ARM Cortex-A9** use **MESI**
- others?

Cache coherence is an implementation detail (not part of the ISA). Hence programmers do not require to understand it to develop correct programs.

But it can impact performance optimizations!

CACHE PROTOCOL OPTIMIZATIONS

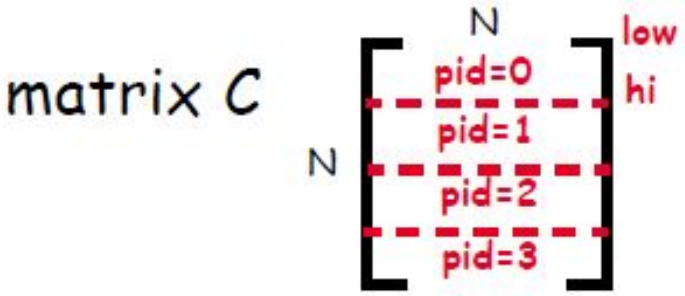
- **producer-consumer sharing; e.g. one processor writes and others (say N proc.) read the same data**
  - results in invalidation followed by N read misses
  - can do better by letting one read miss bring block into all N caches – read *snarfing* (or *broadcast*): load MB if data is in cache in state i
- **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 a method to detect migratory sharing



EXAMPLE: MATRIX MULTIPLY + SUM

SHARED-MEMORY PROGRAM

```
      /* A, B, C, BAR, LV and sum are shared
      /* All other variables are private
1a    low = get_thread_num()*N/nproc; //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);
```



UPDATE-BASED PROTOCOLS

- **update (instead of invalidate) block copies**
  - when sharing is high (eg producer-consumer) invalidate is inefficient
  - if every written value is consumed, update protocols will outperform invalidate-based protocols (shorter latency and less bandwidth)
- **introduce a new bus transaction: BusUpdate**
  - BusUpdate is cheaper than BusRd: only updated words are transferred, not full memory block (eg. 8 bytes vs 64 bytes)
- However if write runs are long (=many local writes before consumer reads) update is less efficient than invalidate (particularly in cc-NUMA!)

Bandwidth trade-off (invalidate vs update):

- write-run of length **N**
- invalidate: **Bandwidth** = 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)

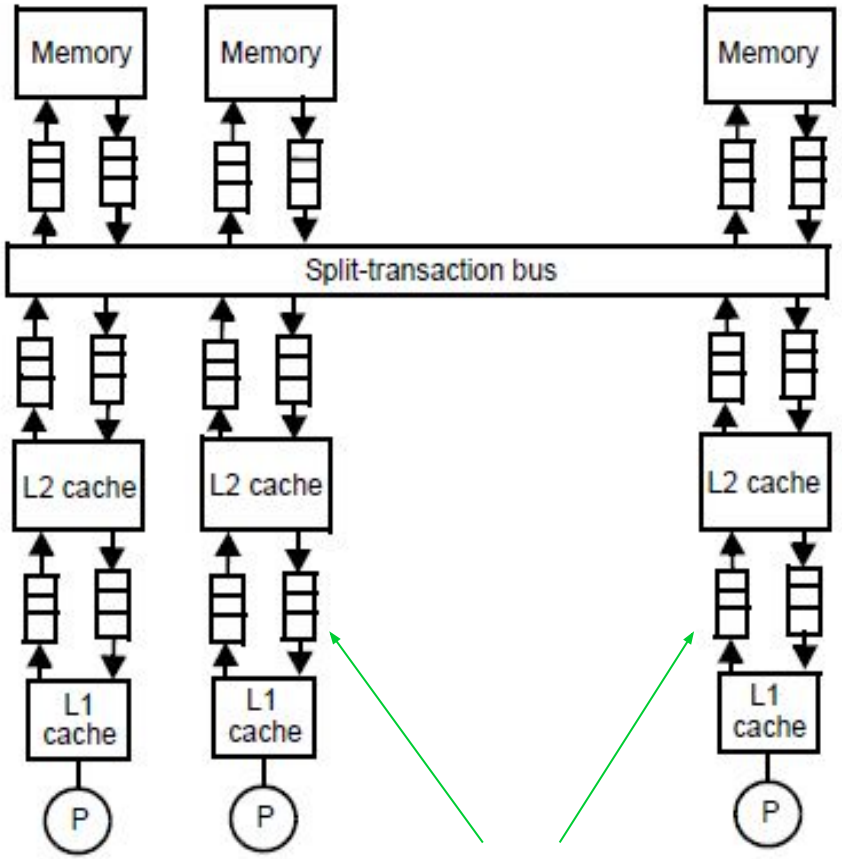
MULTIPHASE CACHE PROTOCOLS

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



FIFO Request Buffers
(smooth speed differences, e.g. bursty traffic)

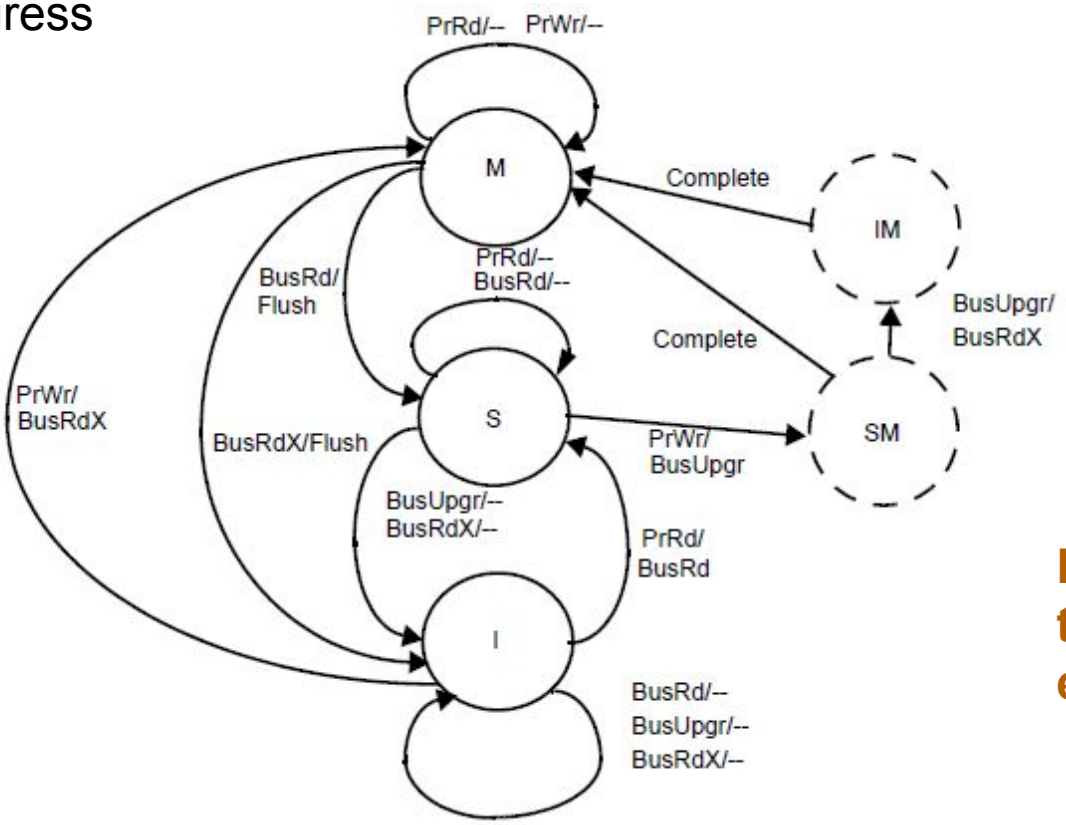
TRANSIENT (NON-ATOMIC) CACHE STATES

Using ‘stable’ states (e.g. M S I) as an abstraction is too high-level to resolve races in cache protocols. Transitions cannot happen atomically!

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. They are transient (as opposed to stable) since they exist only as long as the current transaction is in progress



TRANSIENT STATES
S → M
upon invalidation, need to update request from BusUpgr to BusRdX

Example of adding transient states. Not exhaustive!

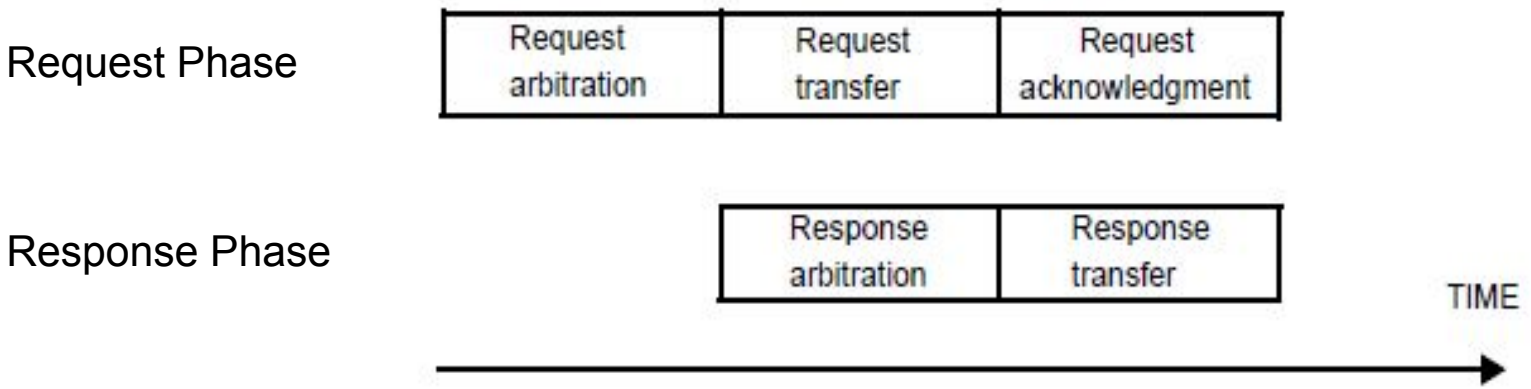
SPLIT-TRANSACTION BUS

- **Split-transaction bus**: pipelines a sequence of phases in a bus transaction; e.g. request and response

- must divide transaction into subtransactions: **adds additional latency** (due to arbitration of each subtransaction) **but increases bandwidth**:
  - bus no longer occupied during full transaction
  - can overlap multiple bus transactions
  - **trade-off latency vs bandwidth**

- must balance pipeline stages to maximize throughput

- separation between request/response arbitration. Use address (or unique transaction ID) to match flushed blocks with requests.



MSI

BusRd, BusUpgr, BusRdX

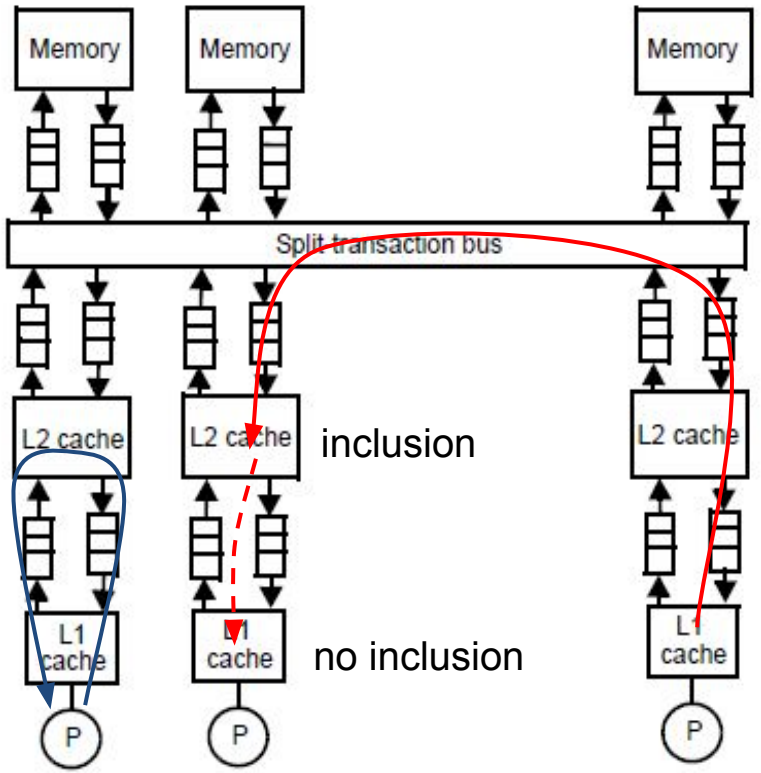
Flush: (Memory → Cache)
or (Cache → Cache)

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 if inclusion is maintained

If not maintained, snoop result cannot be reported until first level cache has been checked.
Increases contention on FLC accesses



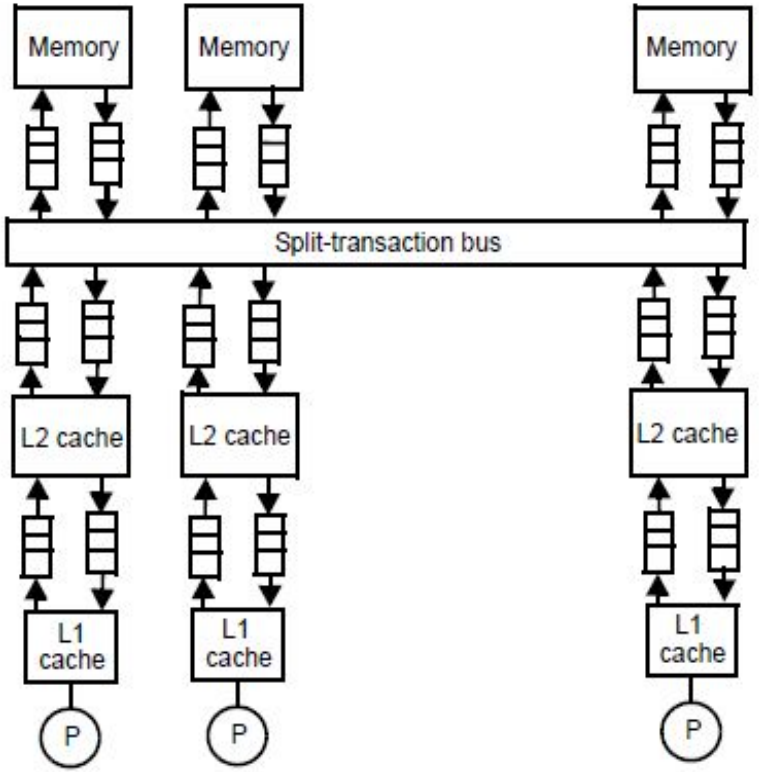
MULTI-LEVEL CACHE ISSUES

inclusion is not always easy to maintain

- to maintain inclusion need to evict a block at level 1 when the block is evicted at level 2 (called “inclusion victims”)
- checking L1 on each L2 replacement may lock out processor

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 (adds latency)
- if level 1 is write-through and inclusion is maintained, level 2 can always respond to miss requests from other processors; can reduce overhead, but increases write traffic



SUMMARY

- **PARALLEL PROGRAMMING MODELS**
  - shared memory vs message passing

- **BUS-BASED SHARED MEMORY SYSTEMS (I)**
  - coherence
  - snoopy cache protocols (VI, MSI, MESI, MOESI)
  - optimizations, updates,
  - multi-level caches