Parallel Processors

- 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 languages 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

Flynn's classification of parallel computers

Classify based on whether the architecture has a single instruction stream (SI), or multiple instruction stream (MI) and whether the machine Processes multiple data (MD) elements, or, single data elements (SD).

Single Instruction Single Data (SISD)
Uniprocessor, superscalar, vliw

Single Instruction Multiple Data (SIMD)
SIMD processors, GPUs

Multiple Instruction Multiple Data (MIMD)
"Multi-core"

Multiple Instruction Single Data (MISD)
No known examples

Data Versus 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

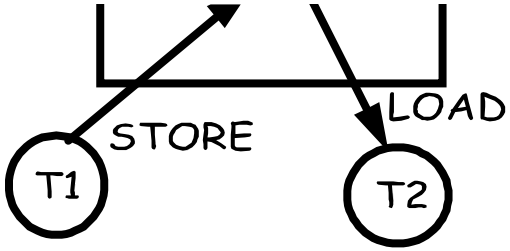
- **Combine both: function + data parallelism**

Amdahl's Law and Parallel Computers

- Amdahl's Law (FracX: original % to be speed up)
  Speedup = 1 / [(FracX/SpeedupX + (1-FracX)]
- A portion is sequential => limits parallel speedup
  - **Speedup <= 1/ (1-FracX)**
- Ex. What fraction sequential to get 80X speedup from 100 processors? Assume either 1 processor or 100 fully used
- 80 = 1 / [(FracX/100 + (1-FracX)]
- 0.8*FracX + 80*(1-FracX) = 80 - 79.2*FracX = 1
- FracX = (80-1)/79.2 = 0.9975
- Only 0.25% sequential!

Inter-PE 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

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

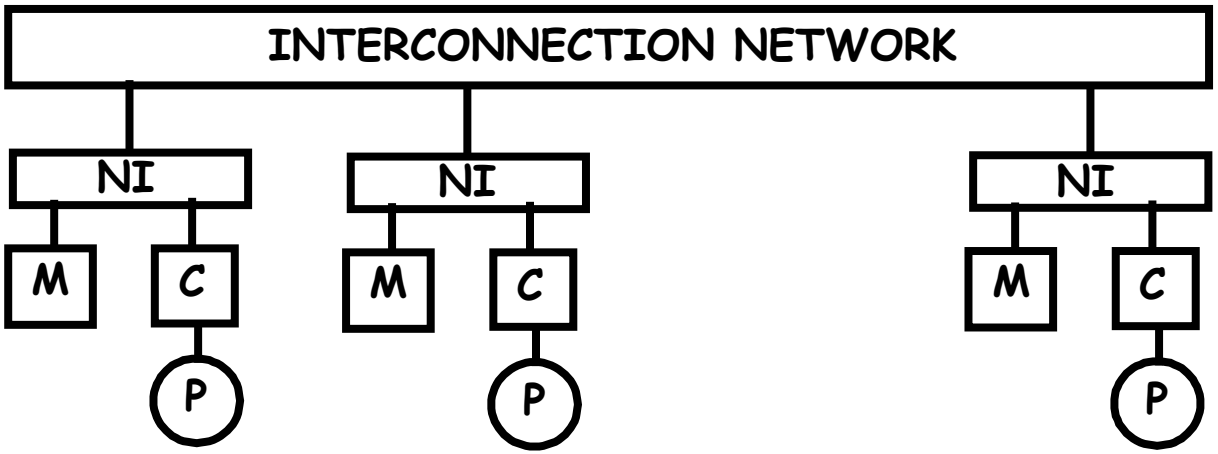
  - The programmer's expectation is that, whatever the order of execution of the two statements is, the final result will be that Sum is incremented by 2
  - However program statements are not executed in an atomic fashion.

- 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

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

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

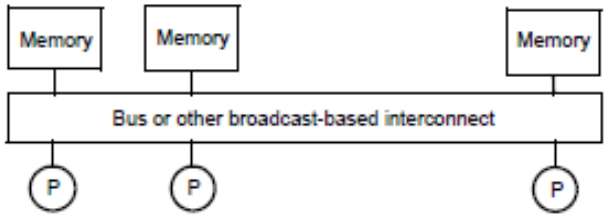
Bus-based Shared Memory Systems

- **Multiprocessor Cache Organizations**

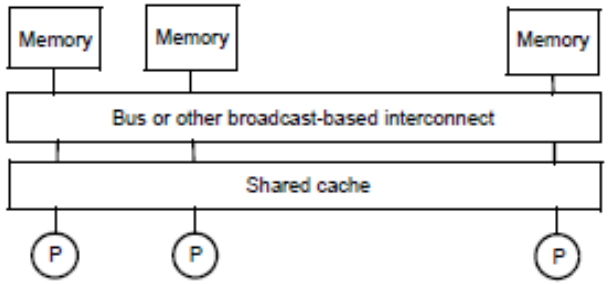
- **Snoopy Cache Protocols**
  - A Simple Snoopy Cache Protocol
  - Design Space Of Snoopy Cache Protocols
  - Protocol Variations
  - Multi-phase Cache Protocols

- **Classification Of Communication Events**

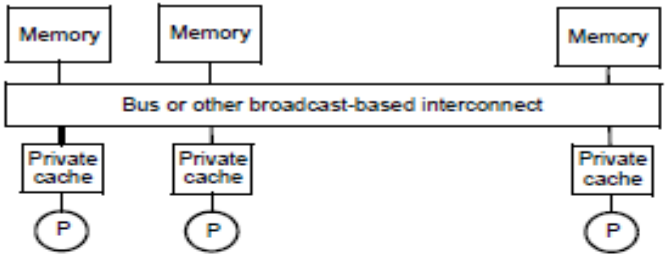
CACHE ORGANIZATIONS



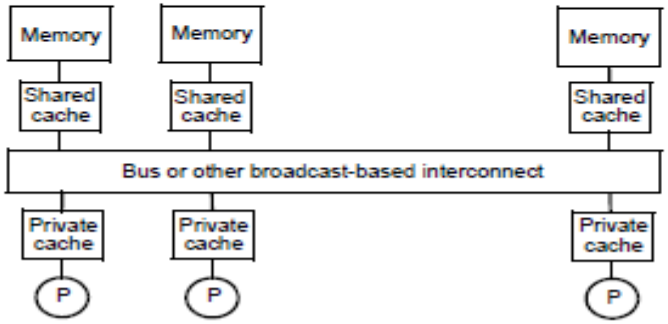
(a) Dance-hall multiprocessor architecture or SMP



(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 In (C)

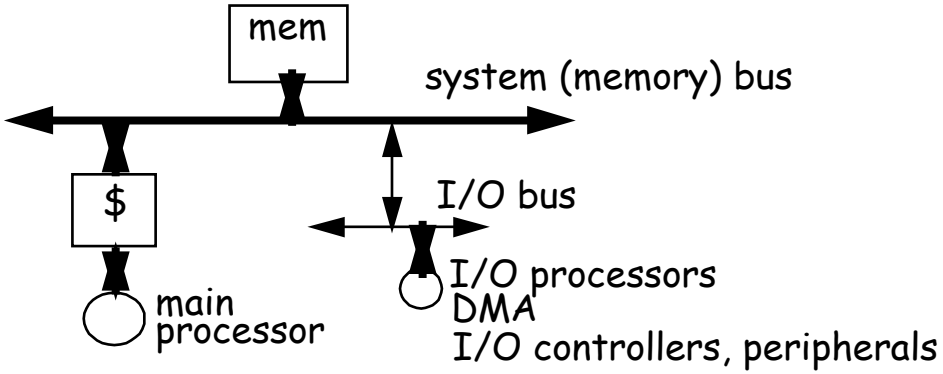
(C) Private L1 Caches

(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 management 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

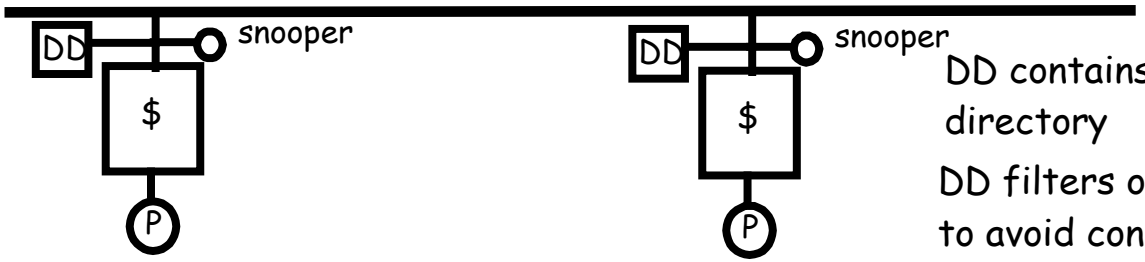
- **In multiprocessors the coherence problem is pervasive, performance critical and software is not informed**
  - Sharing of data, thread migration and I/O
  - Communication is implicit
  - Thus the hardware must solve the problem.

What Does Coherency Mean in MPs

- Informally:
  - **"Any read must return the most recent write"**
  - **Too strict and too difficult to implement**
- Better:
  - **"Any write must eventually be seen by a read"**
  - **All writes are seen in proper order ("serialization")**
- Two rules to ensure this:
  - **"If P writes x and P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart"**
  - **Writes to a single location are serialized: seen in one order**
    - Latest write will be seen
    - Otherwise could see writes in illogical order (could see older value after a newer value)

Snooping-based cache coherence

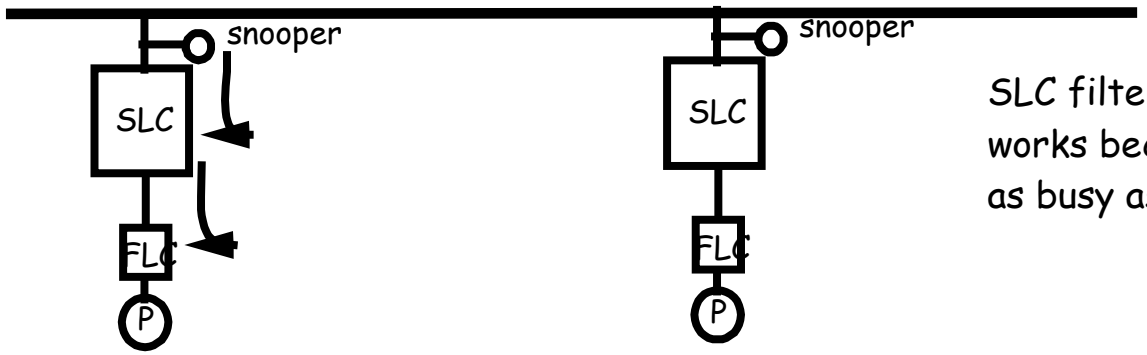
- **Basic idea**
  - Transactions on the bus are "visible" to all processors
  - Bus interface can "snoop" (monitor) the bus traffic and take action when required
  - To take action the "snooper" must check the status of the cache
- **Dual directory**



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
  -

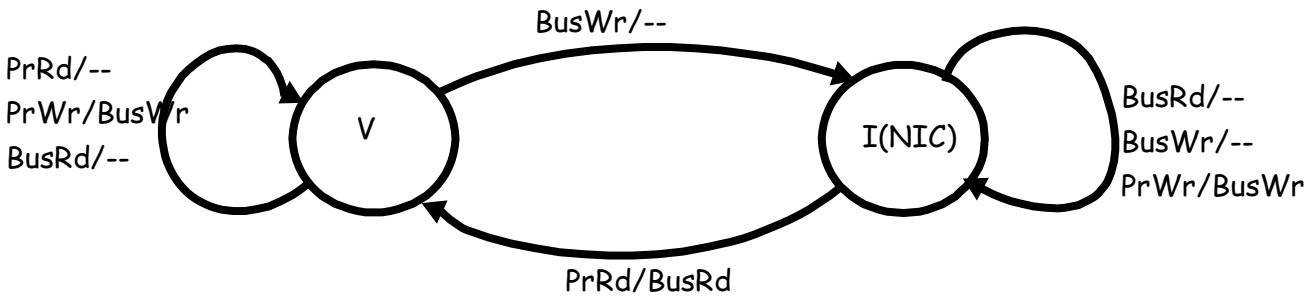


SLC filters out bus request works because SLC is not as busy as FLC

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



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

Potential HW Coherency Solutions

- Snooping Solution (Snoopy Bus):
  - **Send all requests for data to all processors**
  - **Processors snoop to see if they have a copy and respond accordingly**
  - **Requires broadcast, since caching information is at processors**
  - **Works well with bus (natural broadcast medium)**
  - **Dominates for small scale machines (most of the market)**
- Directory-Based Schemes
- Keep track of what is being shared in 1 centralized place (logically)
  - **Distributed memory => distributed directory for scalability (avoids bottlenecks)**
  - **Send point-to-point requests to processors via network**
  - **Scales better than Snooping**
  - **Actually existed BEFORE Snooping-based schemes**

Basic Snoopy Protocols

- Write Invalidate Protocol:
  - **Multiple readers, single writer**
  - **Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies**
  - **Read Miss:**
    - Write-through: memory is always up-to-date
    - Write-back: snoop in caches to find most recent copy
- Write Broadcast Protocol (typically write through):
  - **Write to shared data: broadcast on bus, processors snoop, and *update* any copies**
  - **Read miss: memory is always up-to-date**
- Write serialization: bus serializes requests!
  - **Bus is single point of arbitration**

Basic Snoopy Protocols

- Write Invalidate versus Broadcast:
  - **Invalidate requires one transaction per write-run**
  - **Invalidate uses spatial locality: one transaction per block**
  - **Broadcast has lower latency between write and read**

Snooping Cache Variations

Basic Protocol	Berkeley Protocol Owned Exclusive	Illinois Protocol Private Dirty	MESI Protocol Modified (private,!=Memory)
Exclusive	Owned Shared	Private Clean	exclusive (private,=Memory)
Shared	Shared	Shared	Shared (shared,=Memory)
Invalid	Invalid	Invalid	Invalid

Owner can update via bus invalidate operation
Owner must write back when replaced in cache

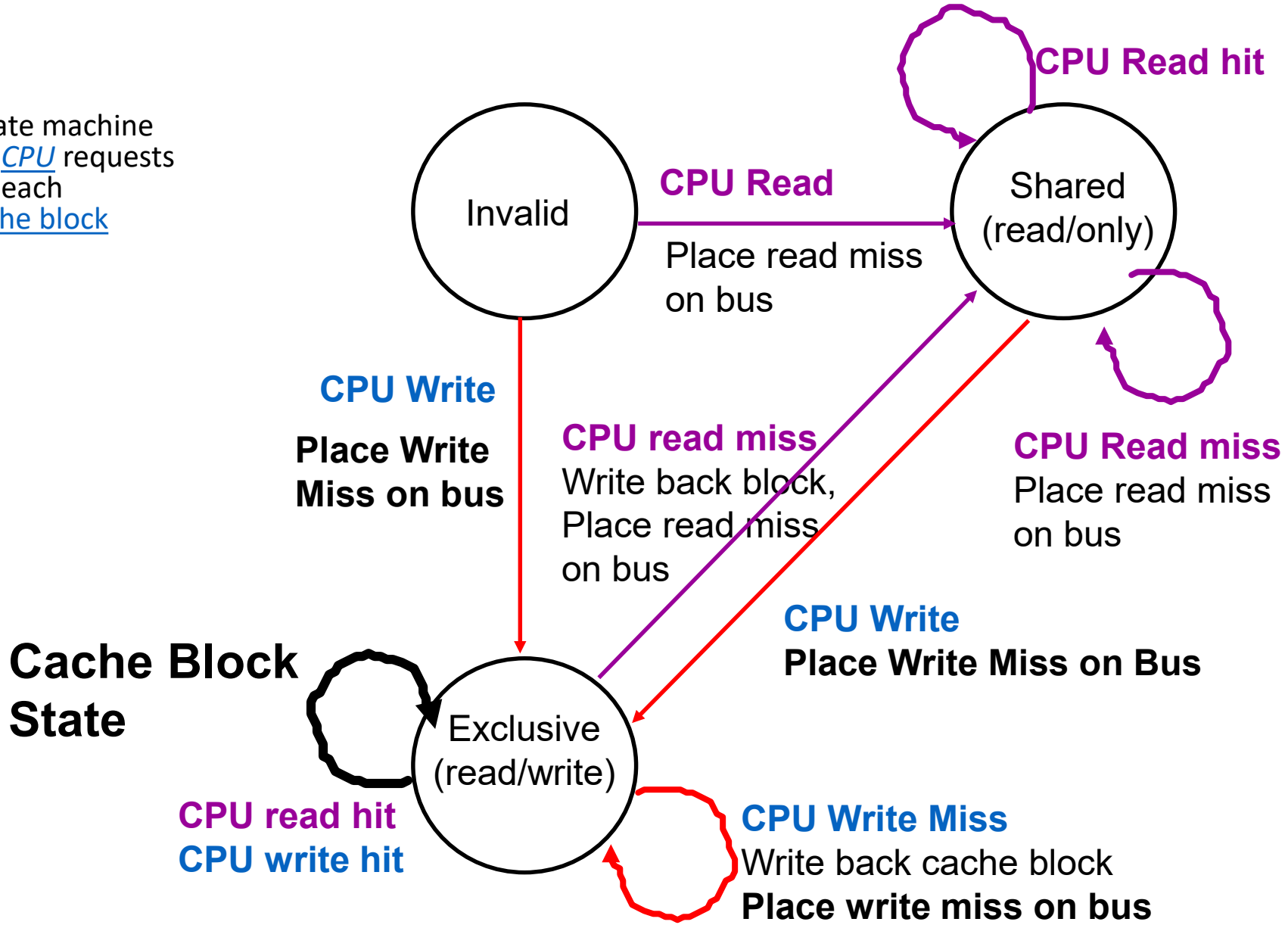
If read sourced from memory, then Private Clean
if read sourced from other cache, then Shared
Can write in cache if held private clean or dirty

An Example Snoopy Protocol

- Invalidation protocol, write-back cache
- Each block of memory is in one state:
  - **Clean in all caches and up-to-date in memory (Shared)**
  - **OR Dirty in exactly one cache (Exclusive)**
  - **OR Not in any caches**
- Each cache block is in one state (track these):
  - **Shared : block can be read**
  - **OR Exclusive : cache has only copy, its writeable, and dirty**
  - **OR Invalid : block contains no data**
- Read misses: cause all caches to snoop bus
- Writes to clean line are treated as misses

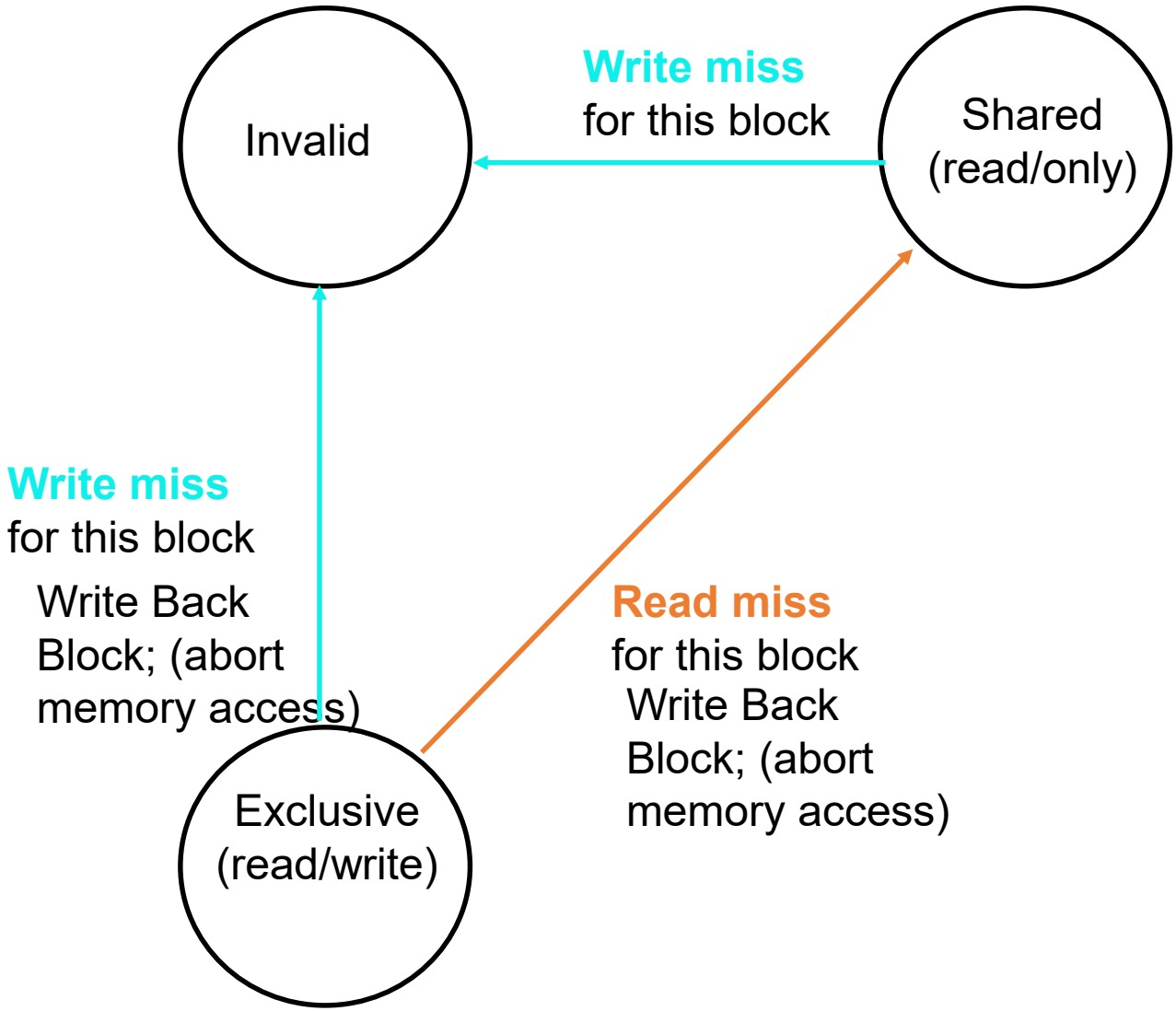
Snoopy-Cache State Machine-I

•State machine for *CPU* requests for each cache block



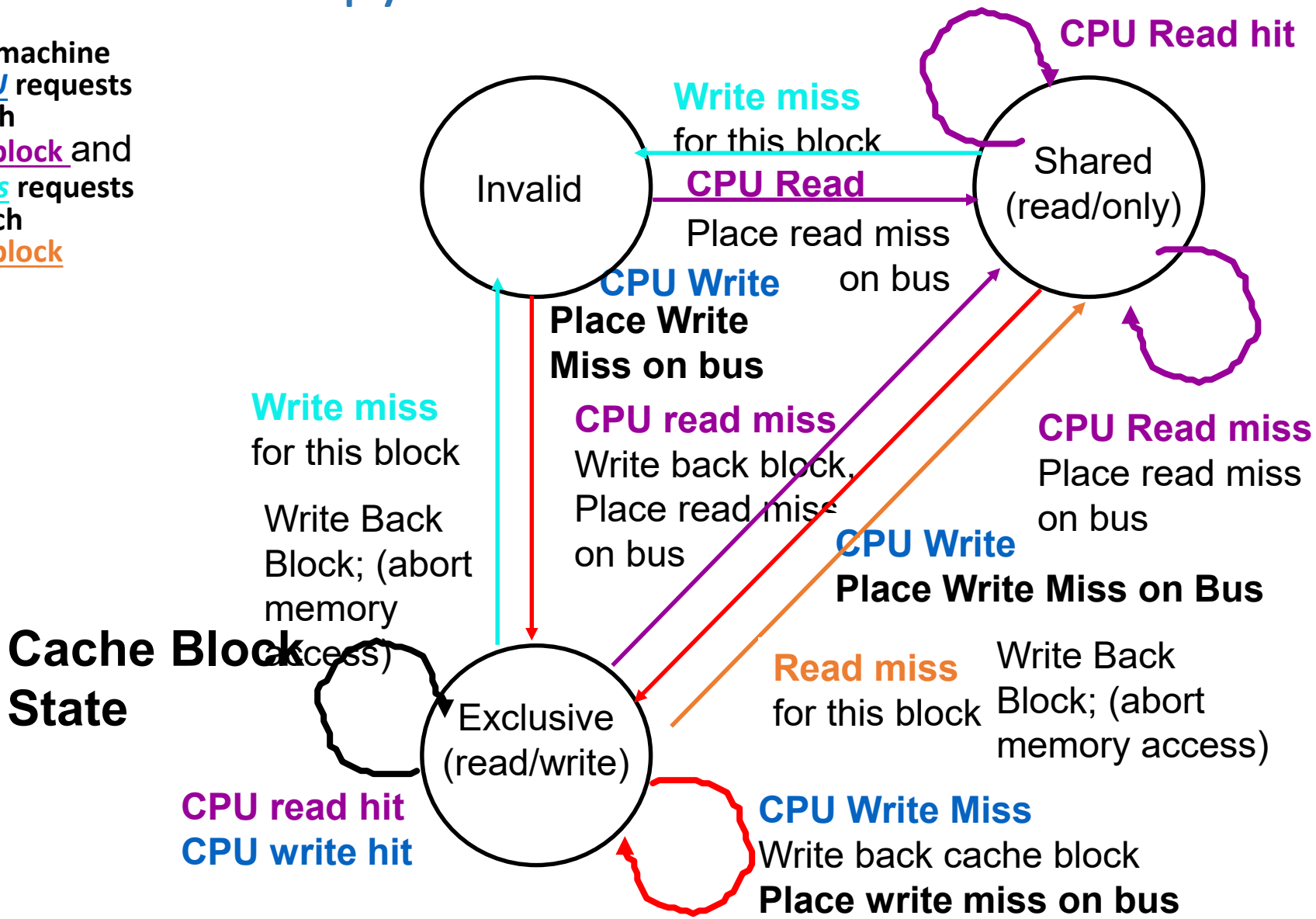
Snoopy-Cache State Machine-II

•State machine for *bus* requests for each cache block



Snoopy-Cache State Machine-III

- State machine for *CPU* requests for each cache block and for *bus* requests for each cache block



Implementing Snooping Caches

- **Multiple processors must be on bus, access to both addresses and data**
- **Add a few new commands to perform coherency, in addition to read and write**
- **Processors continuously snoop on address bus**
  - If address matches tag, either invalidate or update
- **Since every bus transaction checks cache tags, could interfere with CPU just to check:**
  - solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache
    - block size, associatively of L2 affects L1

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			COLD MISS
2		R_B		COLD MISS
3			R_C	COLD MISS
4			R_D (evict block B1)	COLD MISS
5	W_A			
6		R_A		TRUE SHARING MISS
7	W_B			
8		R_A		FALSE SHARING MISS
9			R_C	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

Snoopy Coherence Protocols

- **Complications for the basic MSI protocol:**
  - Operations are not atomic
    - E.g. detect miss, acquire bus, receive a response
    - Creates possibility of deadlock and races
    - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate

- **Extensions:**
  - Add exclusive state to indicate clean block in only one cache (MESI protocol)
    - Prevents needing to write invalidate on a write
  - Owned state

MESI Protocol

The letters in the acronym MESI represent four exclusive states that a cache line can be marked with (encoded using two additional bits):

Modified (M)

The cache line is present only in the current cache, and is dirty - it has been modified (M state) from the value in main memory. The cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state. The write-back changes the line to the Shared state(S).

Exclusive (E)

The cache line is present only in the current cache, but is clean - it matches main memory. It may be changed to the Shared state at any time, in response to a read request. Alternatively, it may be changed to the Modified state when writing to it.

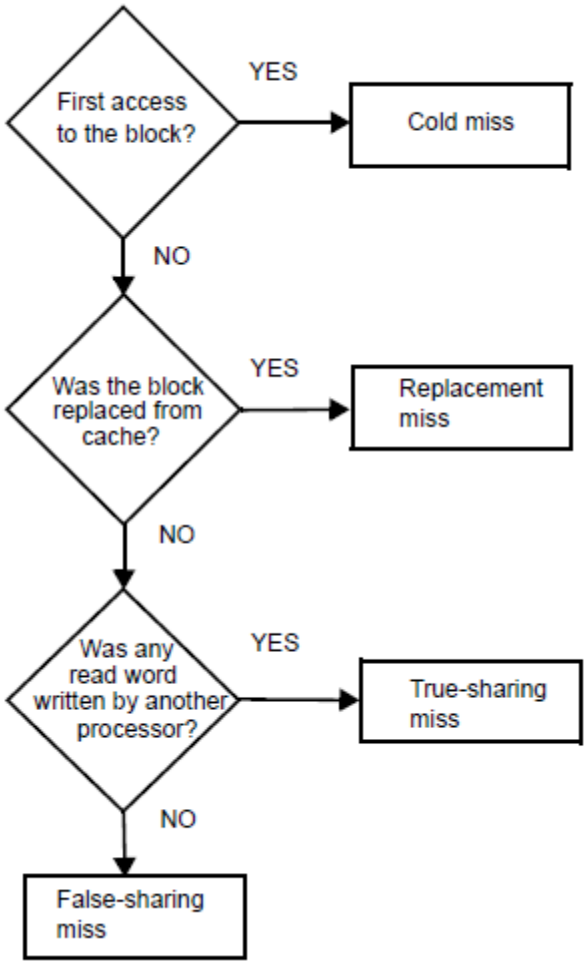
Shared (S)

Indicates that this cache line may be stored in other caches of the machine and is clean - it matches the main memory. The line may be discarded (changed to the Invalid state) at any time.

Invalid (I)

Indicates that this cache line is invalid (unused).

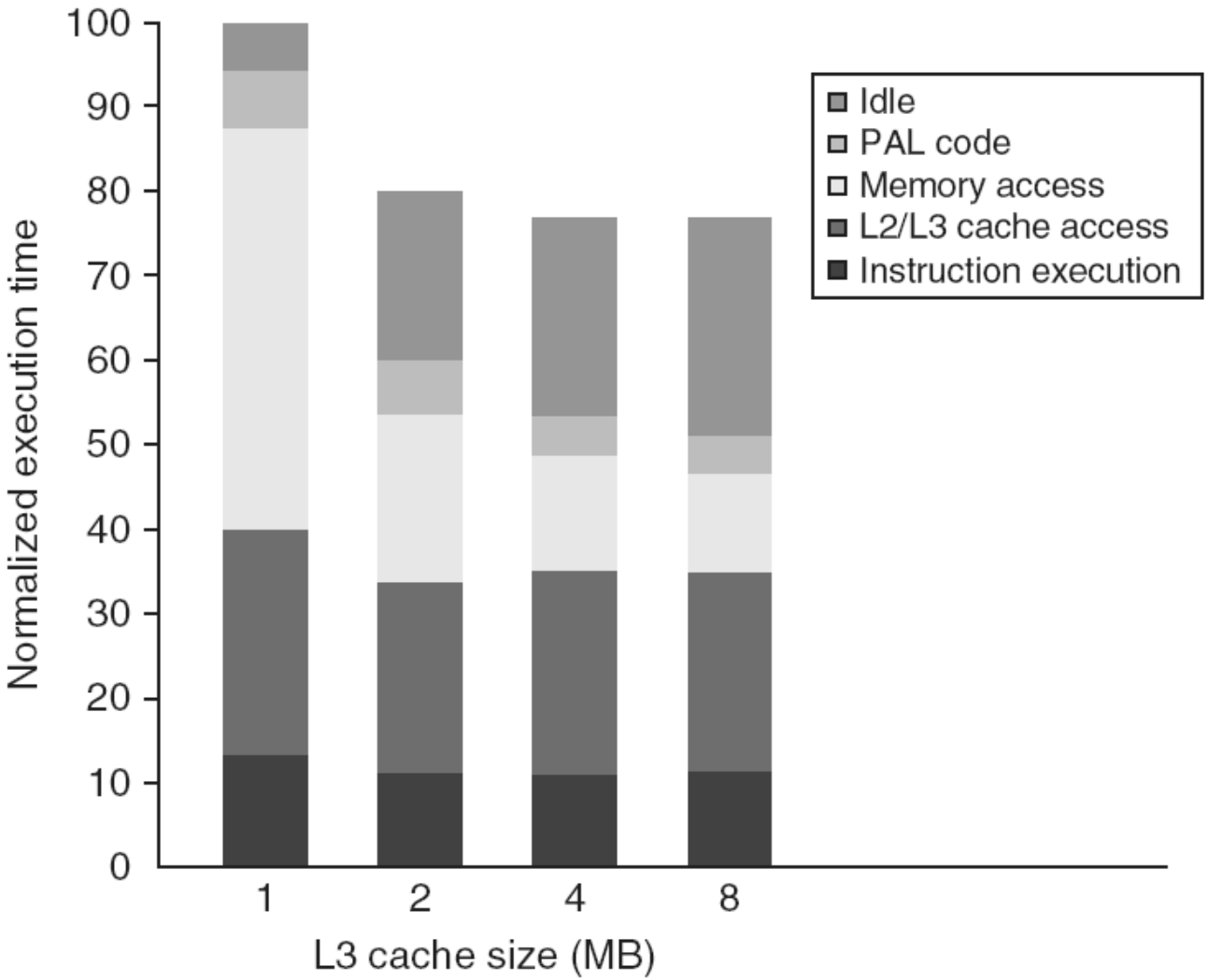
CLASSIFICATION OF MISSES



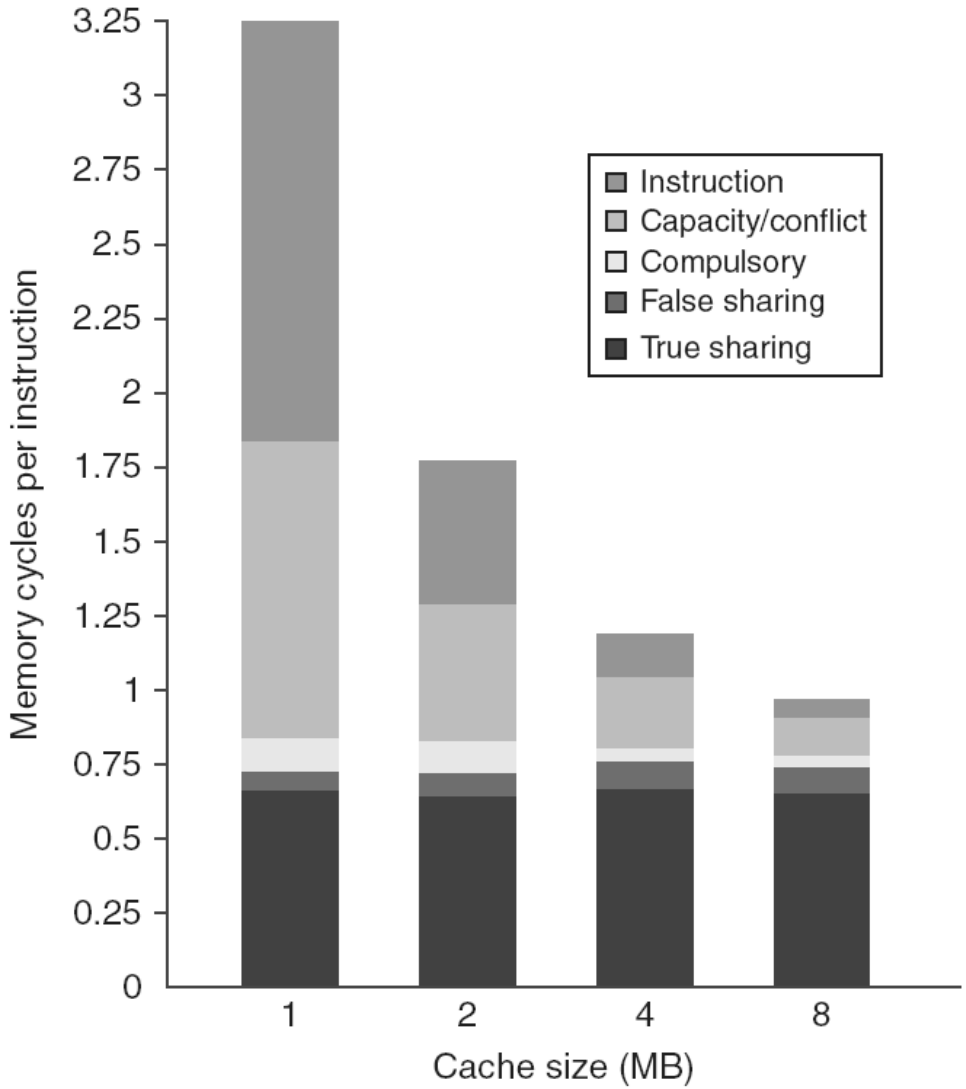
Performance

- **Coherence influences cache miss rate**
  - Coherence misses
    - True sharing misses
      - Write to shared block (transmission of invalidation)
      - Read an invalidated block
    - False sharing misses
      - Read an unmodified word in an invalidated block

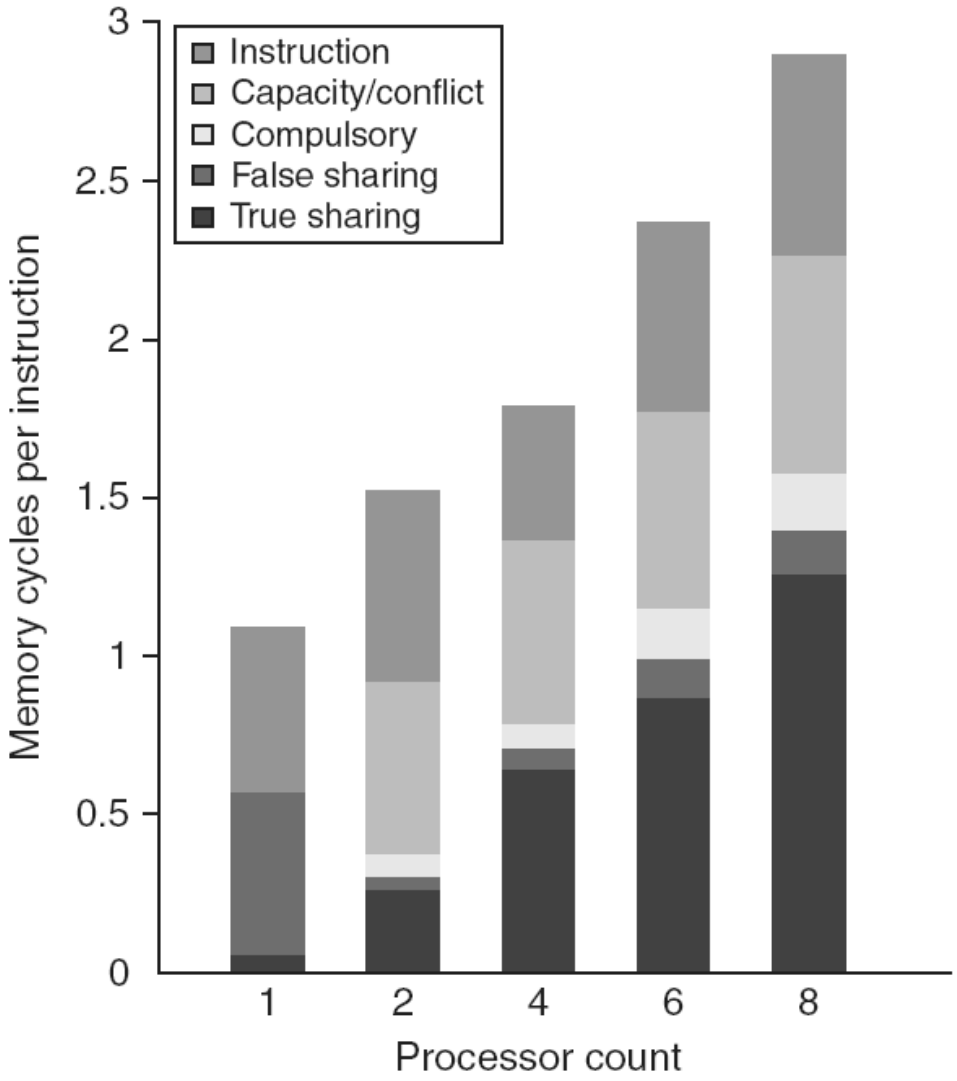
Performance Study: Commercial Workload



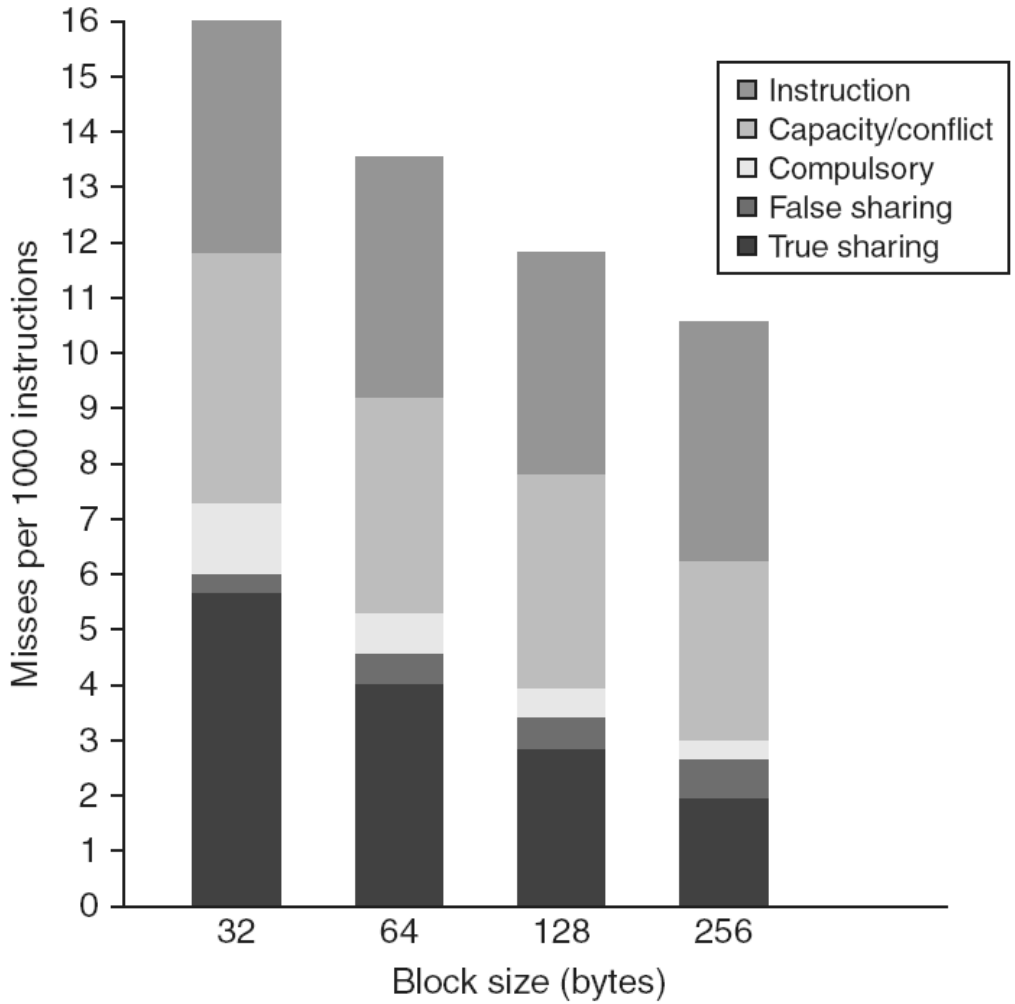
Performance Study: Commercial Workload



Performance Study: Commercial Workload

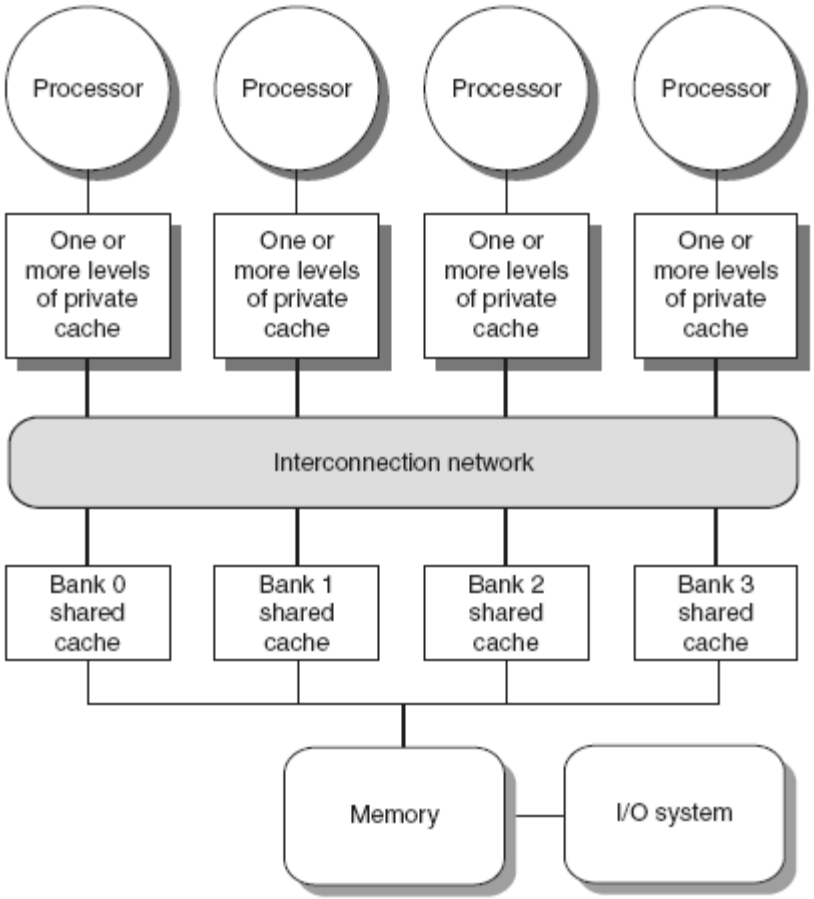


Performance Study: Commercial Workload



Coherence Protocols: Extensions

- **Shared memory bus and snooping bandwidth is bottleneck for scaling symmetric multiprocessors**
  - Duplicating tags
  - Place directory in outermost cache
  - Use crossbars or point-to-point networks with banked memory

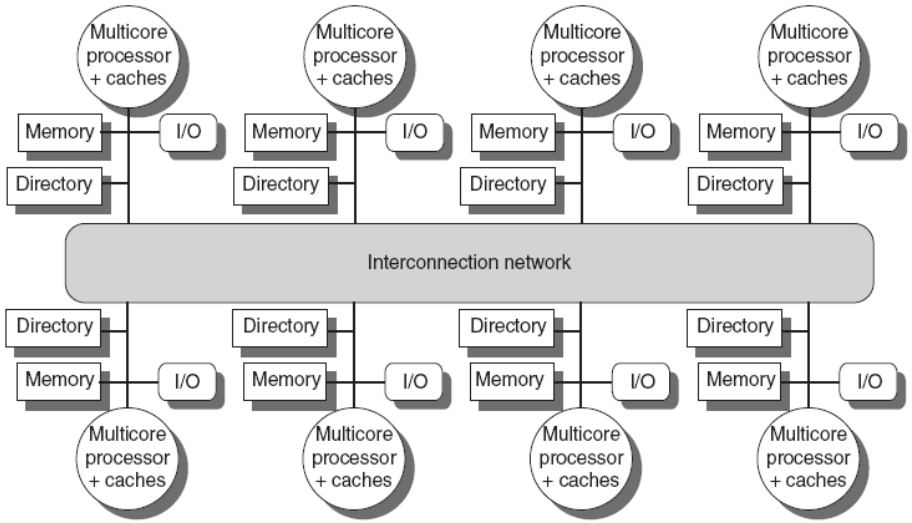


Coherence Protocols

- **AMD Opteron:**
  - Memory directly connected to each multicore chip in NUMA-like organization
  - Implement coherence protocol using point-to-point links
  - Use explicit acknowledgements to order operations

Directory Protocols

- **Directory keeps track of every block**
  - Which caches have each block
  - Dirty status of each block
- **Implement in shared L3 cache**
  - Keep bit vector of size = # cores for each block in L3
  - Not scalable beyond shared L3
- **Implement in a distributed fashion:**



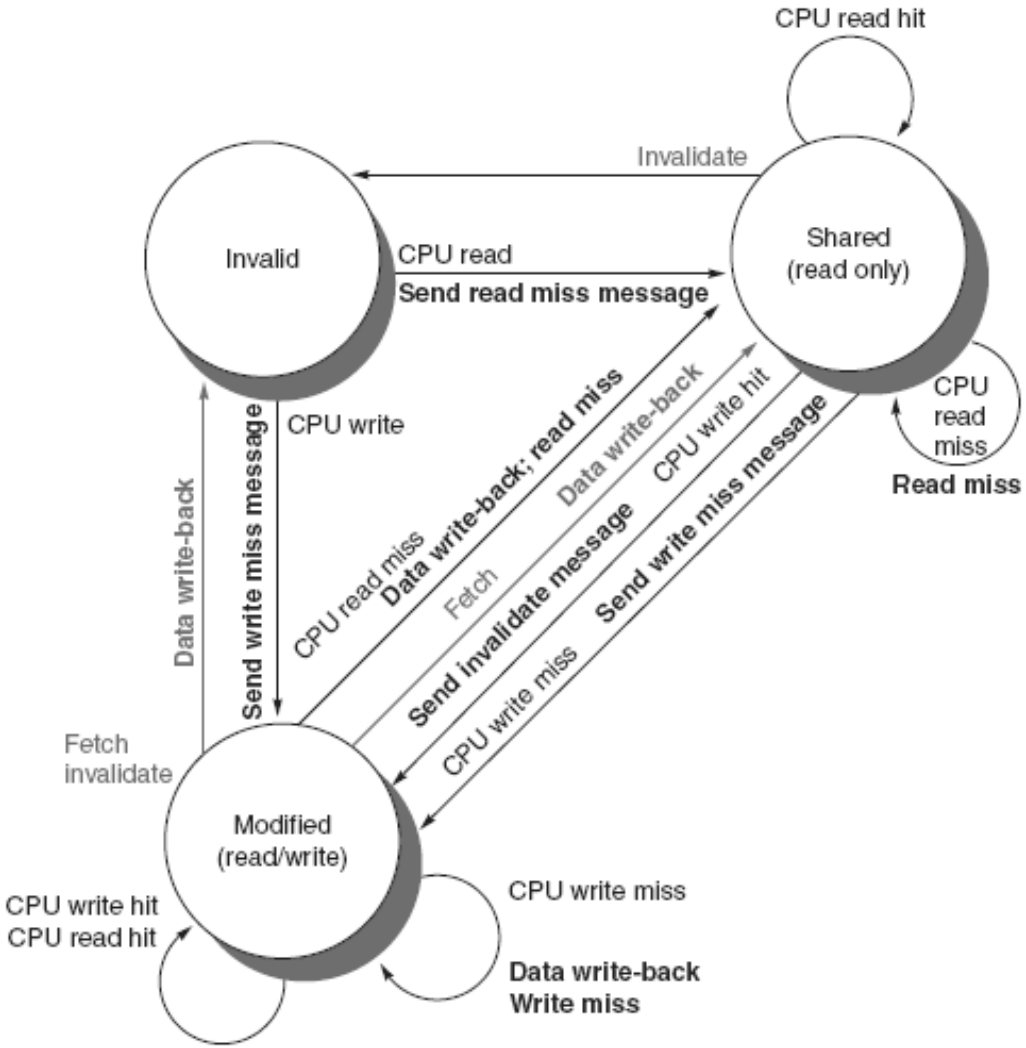
Directory Protocols

- **For each block, maintain state:**
  - Shared
    - One or more nodes have the block cached, value in memory is up-to-date
    - Set of node IDs
  - Uncached
  - Modified
    - Exactly one node has a copy of the cache block, value in memory is out-of-date
    - Owner node ID
- **Directory maintains block states and sends invalidation messages**

Messages

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.

Directory Protocols



Directory Protocols

- **For uncached block:**
  - Read miss
    - Requesting node is sent the requested data and is made the only sharing node, block is now shared
  - Write miss
    - The requesting node is sent the requested data and becomes the sharing node, block is now exclusive
- **For shared block:**
  - Read miss
    - The requesting node is sent the requested data from memory, node is added to sharing set
  - Write miss
    - The requesting node is sent the value, all nodes in the sharing set are sent invalidate messages, sharing set only contains requesting node, block is now exclusive

Directory Protocols

- **For exclusive block:**
  - Read miss
    - The owner is sent a data fetch message, block becomes shared, owner sends data to the directory, data written back to memory, sharers set contains old owner and requestor
  - Data write back
    - Block becomes uncached, sharer set is empty
  - Write miss
    - Message is sent to old owner to invalidate and send the value to the directory, requestor becomes new owner, block remains exclusive

Synchronization

- **Basic building blocks:**
  - Atomic exchange
    - Swaps register with memory location
  - Test-and-set
    - Sets under condition
  - Fetch-and-increment
    - Reads original value from memory and increments it in memory
  - Requires memory read and write in uninterruptable instruction

  - load linked/store conditional
    - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails