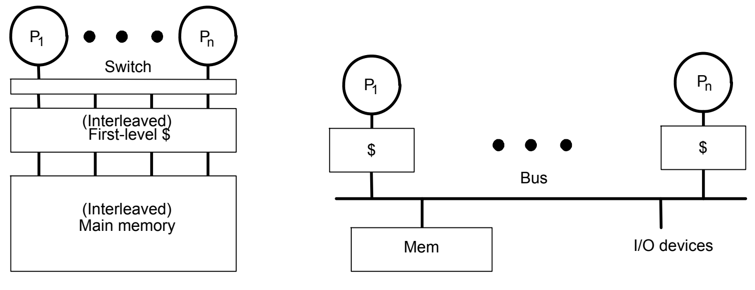
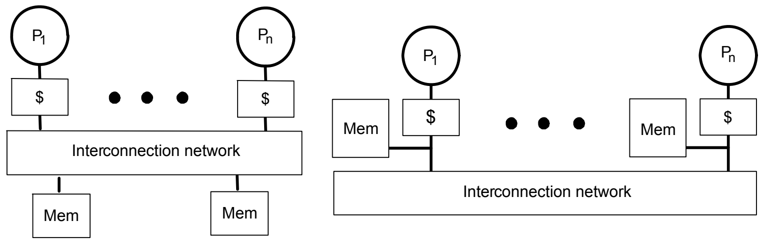
Shared Memory Multiprocessors

Shared Memory Multiprocessors



(a) Shared cache

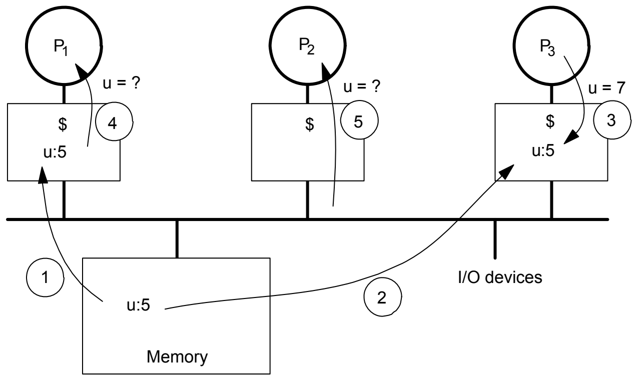
(b) Bus-based shared memory



(c) Dancehall

(d) Distributed-memory

Example Cache Coherence Problem



- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!

A Coherent Memory System: Intuition

- Reading a location should see
  - The latest value written by any process
- On uniprocessors
  - No issues between processes
  - Coherency between DMA operations and processor caches
- Multiprocessors
  - Coherent as if the processes were interleaved on a uniprocessor

Problems with the Intuition

- Value returned by a read should be last value written
  - But, “last” is not well-defined
- In sequential case, last defined in terms of program order, not time
  - Order of operations in the machine language presented to processor
- In parallel case, program order defined within a process
  - Need to make sense of orders across processes

Some Basic Definitions

- *Memory operation*: a single read (load), write (store) or read-modify-write access to a memory location
  - Assumed to execute atomically with respect to each other
- *Issue*: a memory operation issues when it leaves processor’s internal environment and is presented to memory system (cache, buffer …)
- *Perform*: operation appears to have taken place, as far as processor can tell from other memory operations it issues
  - A write performs with respect to the processor when a subsequent read by the processor returns the value of that write or a later write
  - A read perform with respect to the processor when subsequent writes issued by the processor cannot affect the value returned by the read
- In multiprocessors, stay same but replace “the” by “a” processor
  - Also, *complete*: perform with respect to all processors
  - Still need to make sense of order in operations from different processes

Sharpening the Intuition

- Imagine a single shared memory and no caches
  - Every read and write to a location accesses the same physical location
  - Operation completes when it does so
- Memory imposes a serial or total order on operations to the location
  - Operations to the location from a given processor are in program order
  - The order of operations to the location from different processors is some interleaving that preserves the individual program orders
- “Last” now means most recent in a hypothetical serial order that maintains these properties
- For the serial order to be consistent, all processors must see writes to the location in the same order (if they bother to look, i.e. to read)
- Note that the total order is never really constructed in real systems
  - Don’t even want memory, or any hardware, to see all operations
- But program should behave as if some serial order is enforced
  - Order in which things appear to happen, not actually happen

Formal Definition of Coherence

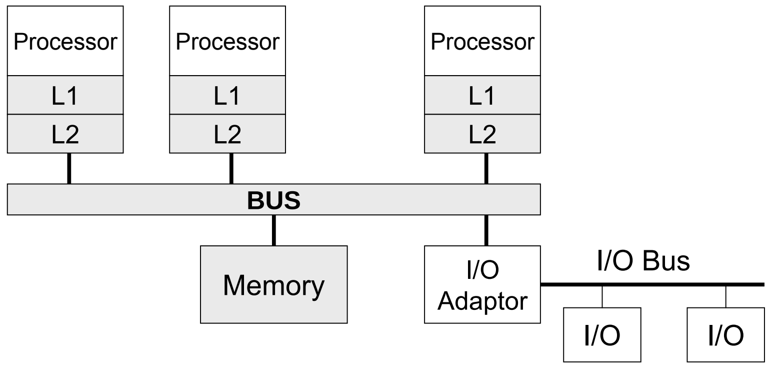
- Results of a program: values returned by its read operations
- A memory system is coherent if the results of any execution of a program are such that each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and in which:
  1. operations issued by any particular process occur in the order issued by that process, and
  2. the value returned by a read is the value written by the last write to that location in the serial order
- Two necessary features:
  - Write propagation: value written must become visible to others
  - Write serialization: writes to location seen in same order by all
    - if I see w1 after w2, you should not see w2 before w1
    - no need for analogous read serialization since reads not visible to others

Potential Hardware Coherency Solutions

- Snooping Solution:
  - 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 a centralized place (logically)
  - Distributed memory ⇒ distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Idea existed before Snooping-based schemes

Bus Snooping Topology

- Memory: centralized with uniform access time (UMA) and bus interconnect
- Early examples: Firefly, Encore, Sequent, …
- Current example: Unisys



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

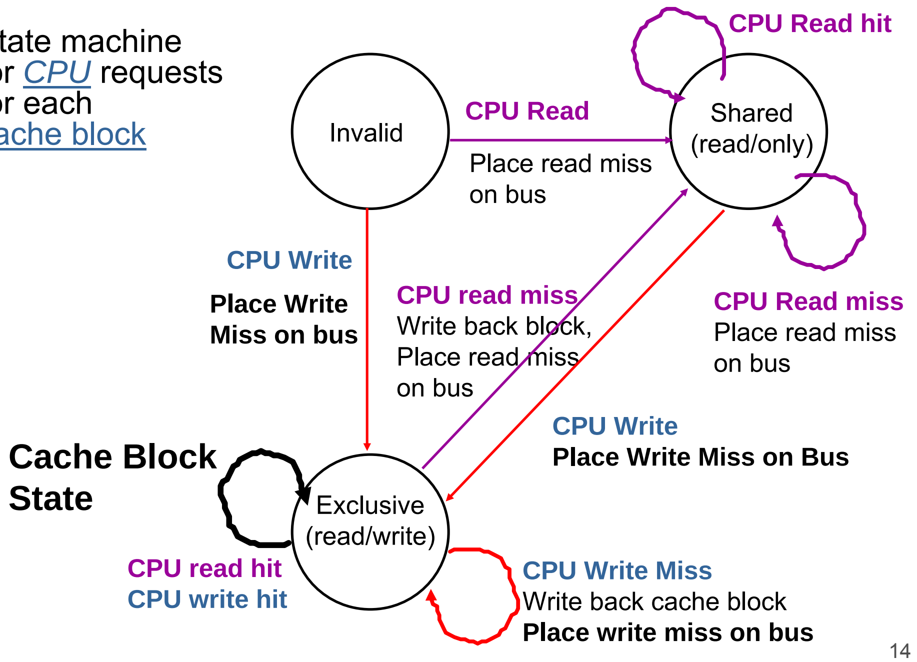
- Write Invalidate versus Broadcast:
  - Invalidate requires one transaction per write-run
  - Invalidate uses spatial locality: one transaction per block
  - Broadcast has lower latency between write and read
- Early write invalidate examples
  - Encore and Sequent systems
- Early broadcast examples
  - DECSRC's firefly
  - Xerox PARC Dragonfly

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

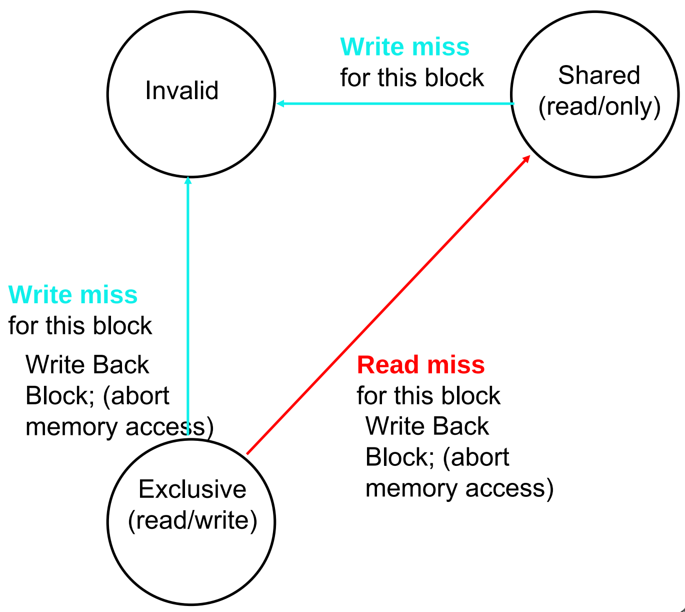
Snoopy-Cache State Machine-I

- State machine for CPU requests for each cache block



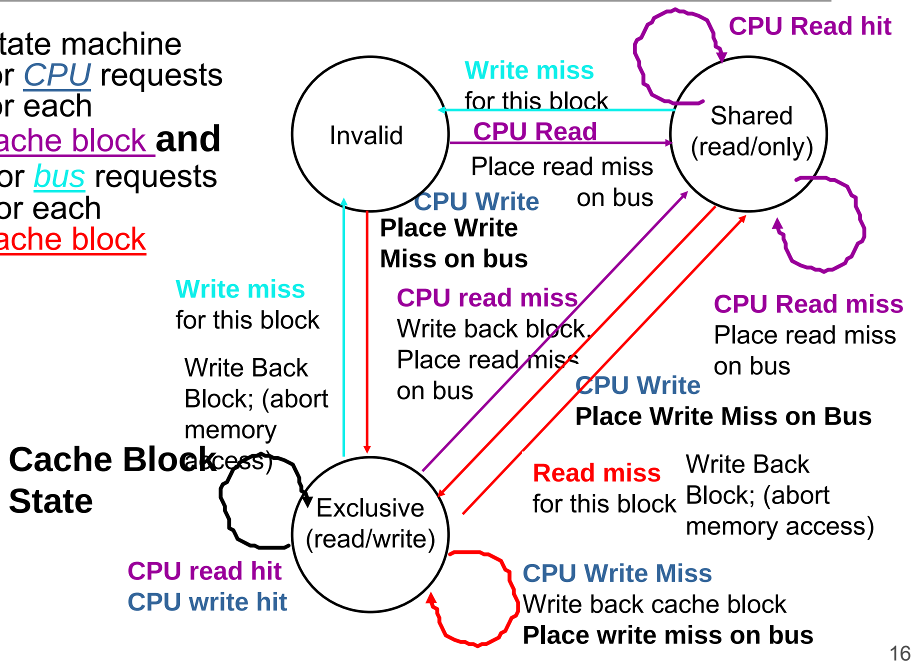
Snoopy-Cache State Machine-II

- State machine for bus requests for each cache block



Snoopy-Cache State Machine-III

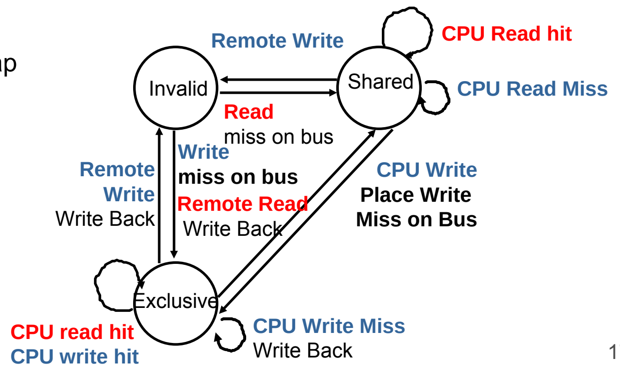
- State machine for CPU requests for each cache block **and** for bus requests for each cache block



Example

	Processor 1			**Processor 2**			**Bus**				**Memory**	
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

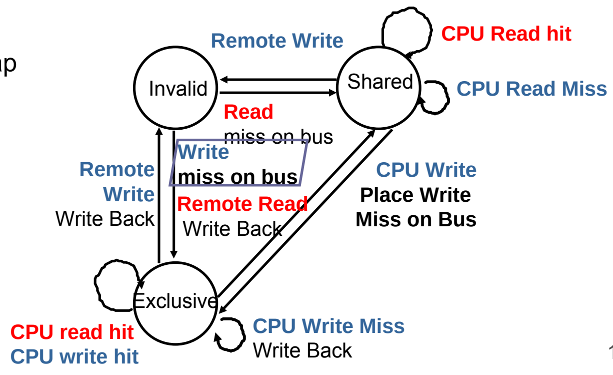
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



Example: Step 1

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

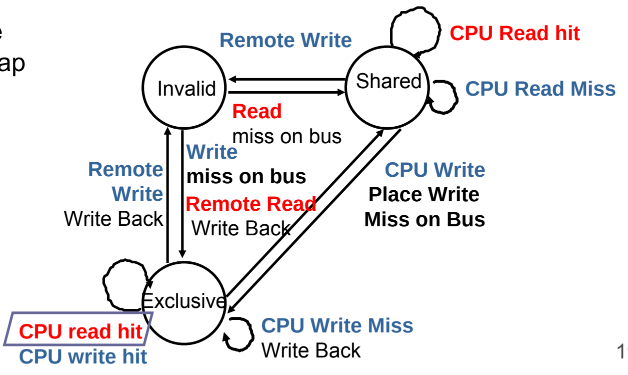
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.
Active arrow =



Example: Step 2

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

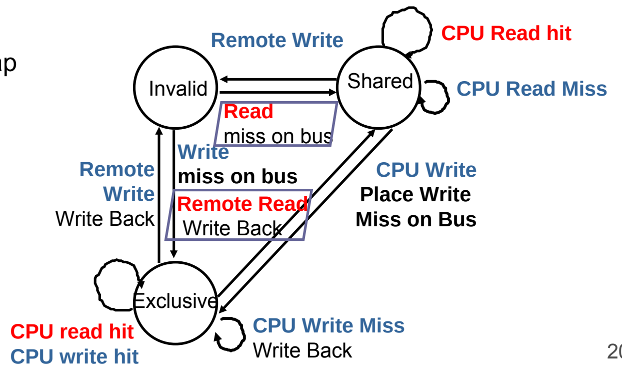
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



Example: Step 3

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1												
P2: Write 40 to A2												

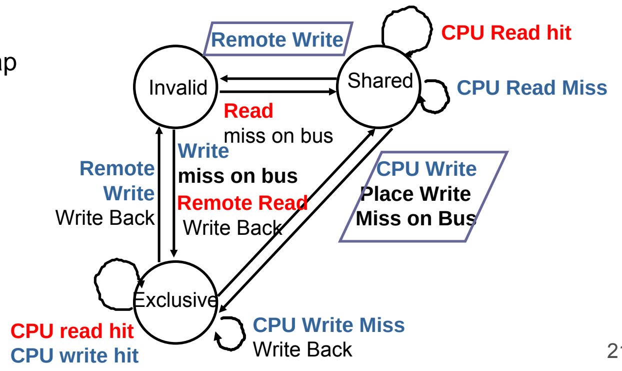
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.



Example: Step 4

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2												

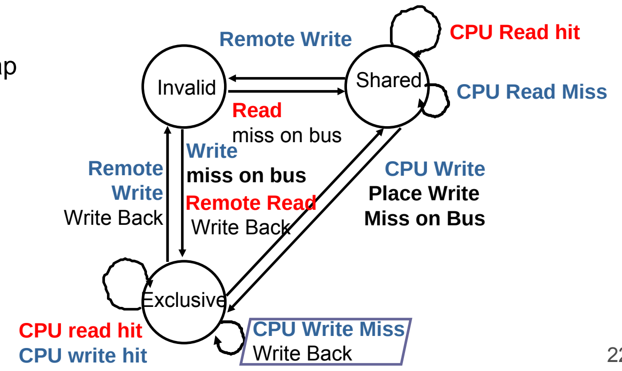
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



Example: Step 5

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2							WrMs	P2	A2		A1	10
				Excl.	A2	40	WrBk	P2	A1	20	A1	20

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



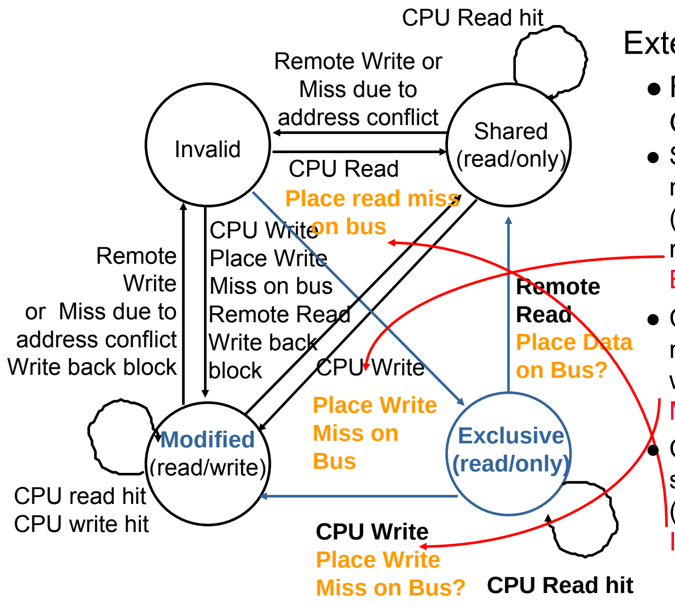
Snooping Cache Variations

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
	Owned Exclusive	Private Dirty	Modfied (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

Snoop Cache Extensions



Extensions:

- Fourth State: Ownership
- Shared-> Modified, need invalidate only (upgrade request), don't read memory
  Berkeley Protocol
- Clean exclusive state (no miss for private data on write)
  MESI Protocol
- Cache supplies data when shared state (no memory access)
  Illinois Protocol

Implementation Complications

- Write Races:
  - Cannot update cache until bus is obtained
    - Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - Arbitrate for bus
    - Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
  - Split transaction bus:
    - Bus transaction is not atomic: can have multiple outstanding transactions for a block
    - Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
    - Must track and prevent multiple misses for one block
- Must support interventions and invalidations

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 cache access:
  - 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, associativity of L2 affects L1

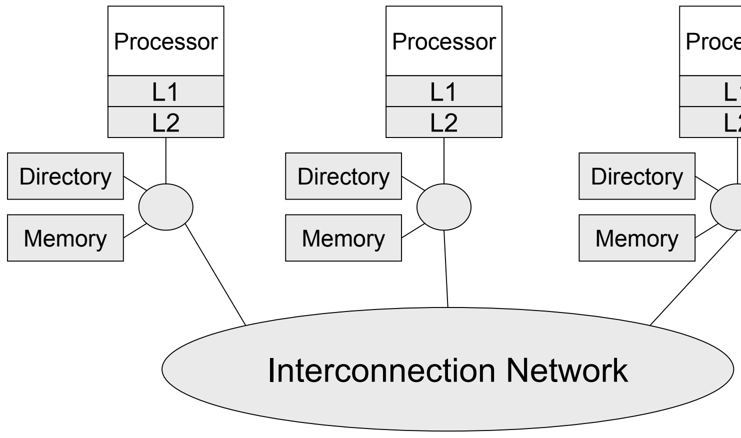
Implementing Snooping Caches

- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write back cache, may have the desired copy and its dirty, so must reply
- Add extra state bit to cache to determine shared or not
- Add 4th state (MESI)

Larger Multiprocessors

- Separate Memory per Processor
- Local or Remote access via memory controller
- Alternative: directory per cache that tracks state of every block in every cache
  - Which caches have a copies of block, dirty vs. clean, ...
- Information per memory block vs. per cache block?
  - PLUS: In memory $\Rightarrow$ simpler protocol (centralized/one location)
  - MINUS: In memory $\Rightarrow$ directory is f(memory size) vs. f(cache size)
- Prevent directory as bottleneck? distribute directory entries with memory, each keeping track of which processors have copies of their blocks

ccNUMA Multiprocessors



Cache Coherent Non-Uniform Memory Access (ccNUMA)

Directory Protocol

- Similar to Snoopy Protocol: Three states
  - **Shared**: ≥ 1 processors have data, memory up-to-date
  - **Uncached** (no processor has it; not valid in any cache)
  - **Exclusive**: 1 processor (owner) has data; memory out-of-date
- Directory must track
  - Cache state
  - Which processors have data when in the shared state
    - Bit vector, 1 if a particular processor has a copy
    - Id and bit vector combination
- Keep it simple:
  - Writes to non-exclusive data ⇒ write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent

Directory Protocol

- No bus and do not want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Terms: typically 3 processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has a copy of a cache block, whether exclusive or shared
- Example messages on next slide:
  P = processor number, A = address

Directory Protocol Messages

- ReadMiss(P, A): from local cache to home directory
  - Processor P reads data at A, makes P the read sharer and requests for data back
- WriteMiss(P, A): from local cache to home directory
  - Processor P writes data at A, makes P the exclusive owner, requests for data back
- Invalidate(P, A): from home directory to remote cache
  - Invalidate a shared copy of A at processor P
- Fetch(P, A): from home directory to remote cache
  - Fetch data at address A from P's cache
- Fetch&Invalidate(P, A): from home directory to remote cache
  - Fetch data at address A from P's cache and invalidate home directory node's cache
- DataReply(P, A): from home directory to remote cache
  - Return data from home directory (read miss)

Implementing a Directory

- Issues
  - Operations are not atomic, why?
  - May have deadlocks, why?
- Solutions
  - Two networks: one for requests and one for replies
  - How can this be helpful?
- Optimizations
  - For read miss or write miss in Exclusive, send data directly to requestor from owner to memory and then from memory to requestor
  - Exclusive node is always the owner, is this difficult?

Synchronization

- Why Synchronize? Need to know when it is safe for different processes to use shared data
- Issues for Synchronization:
  - Uninterruptable instruction to fetch and update memory (atomic operation);
  - User level synchronization operation using this primitive;
  - For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Atomic Memory Instructions

- Atomically swap the contents of a register and a memory location
  - 0 ⇒ synchronization variable is free
  - 1 ⇒ synchronization variable is locked and unavailable
  - Set register to 1 & swap
  - New value in register determines success in getting lock
    - 0 if you succeeded in setting the lock (you were first)
    - 1 if other processor had already claimed access
- Test-and-set
  - Test a value and sets it if the value passes the test
- Fetch-and-op: it returns the value of a memory location and atomically performs an operation
  - 0 => synchronization variable is free

Load Locked and Store Conditional

- Difficult to implement an atomic instruction on a large multiprocessor efficiently
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceeding load) and 0 otherwise
- Example doing atomic swap with LL & SC:

```
try: mov   R3,R4        ; mov exchange value
     ll    R2,0(R1)     ; load linked
     sc    R3,0(R1)     ; store conditional
     beqz  R3,try       ; branch store fails (R3 = 0)
     mov   R4,R2        ; put load value in R4
```

- Example doing fetch & increment with LL & SC:

```
try: ll    R2,0(R1)     ; load linked
     addi  R2,R2,#1     ; increment (OK if reg–reg)
     sc    R2,0(R1)     ; store conditional
     beqz  R2,try       ; branch store fails (R2 = 0)
```

Always Block

```
Acquire(lock) {
  while (!TAS(lock.value))
    Block( lock );
}
```

```
Release(lock) {
  lock.value = 0;
  Unblock( lock );
}
```

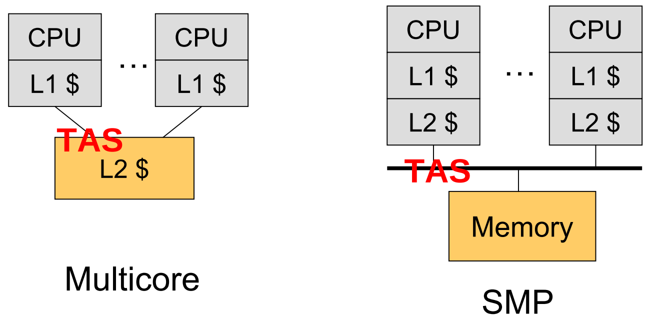
- What are the issues with this approach?

Always Spin

```
Acquire(lock) {
  while (!TAS(lock.value))
    while (lock.value)
      ;
}
```

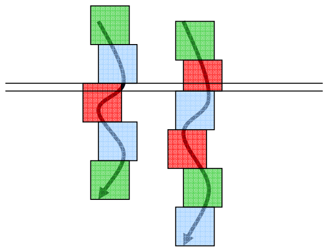
```
Release(lock) {
  lock.value = 0;
}
```

- Two spinning loops in **`Acquire()?`**
- Issues with this approach?



Optimal Algorithms

- What is the optimal solution to spin vs. block?
  - Know the future
  - Exactly when to spin and when to block
- But, we don't know the future
  - There is **no** online optimal algorithm



- Offline optimal algorithm
  - After an execution, we can derive exactly when to block or spin ("what if")
  - Useful to compare against online algorithms

Competitive Algorithms

- An algorithm is c-competitive if for every input sequence σ

$$C_A(\sigma) \le c \times C_{opt}(\sigma) + k$$

  - c is a constant
  - $C_A(\sigma)$ is the cost incurred by algorithm A in processing σ
  - $C_{opt}(\sigma)$ is the cost incurred by the optimal algorithm in processing σ
- What we want is to have c as small as possible
  - Deterministic
  - Randomized

Constant Competitive Algorithms

```
Acquire(lock, N) {
  int i;

  while (!TAS(lock.value))
    for (i = 0; i < N; i++)
      if (!lock.value) break;
  if (lock.value)
    Block(lock);
}
```

- If N is the number of spins equal to the context-switch time, what is the competitive factor of the algorithm?

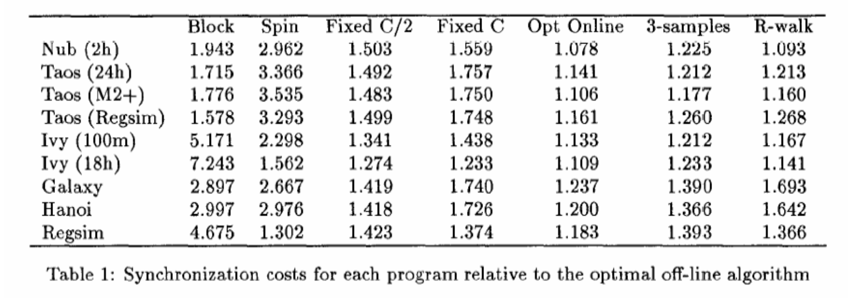
Approximate Optimal Online Algorithms

- Main idea
  - Use past to predict future
- Approach
  - Simplest method is random walk
    - Decrement N by a unit if the last Acquire() blocked
    - Increment N by a unit if the last Acquire() didn't block
  - Recompute N each time for each Acquire() based on some lock-waiting distribution for each lock
- Theoretical results

  $E\ C_A(\sigma (P)) \leq (e/(e-1)) \times E\ C_{opt}(\sigma(P))$

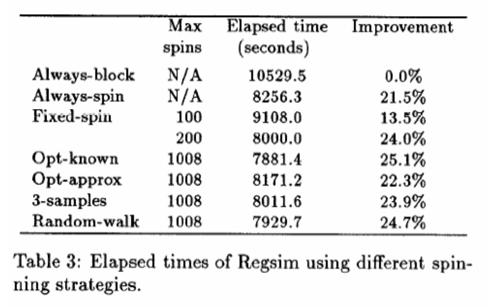
  The competitive factor is about 1.58.

Empirical Results



	Block	Spin	Fixed C/2	Fixed C	Opt Online	3-samples	R-walk
Nub (2h)	1.943	2.962	1.503	1.559	1.078	1.225	1.093
Taos (24h)	1.715	3.366	1.492	1.757	1.141	1.212	1.213
Taos (M2+)	1.776	3.535	1.483	1.750	1.106	1.177	1.160
Taos (Regsim)	1.578	3.293	1.499	1.748	1.161	1.260	1.268
Ivy (100m)	5.171	2.298	1.341	1.438	1.133	1.212	1.167
Ivy (18h)	7.243	1.562	1.274	1.233	1.109	1.233	1.141
Galaxy	2.897	2.667	1.419	1.740	1.237	1.390	1.693
Hanoi	2.997	2.976	1.418	1.726	1.200	1.366	1.642
Regsim	4.675	1.302	1.423	1.374	1.183	1.393	1.366

Table 1: Synchronization costs for each program relative to the optimal off-line algorithm

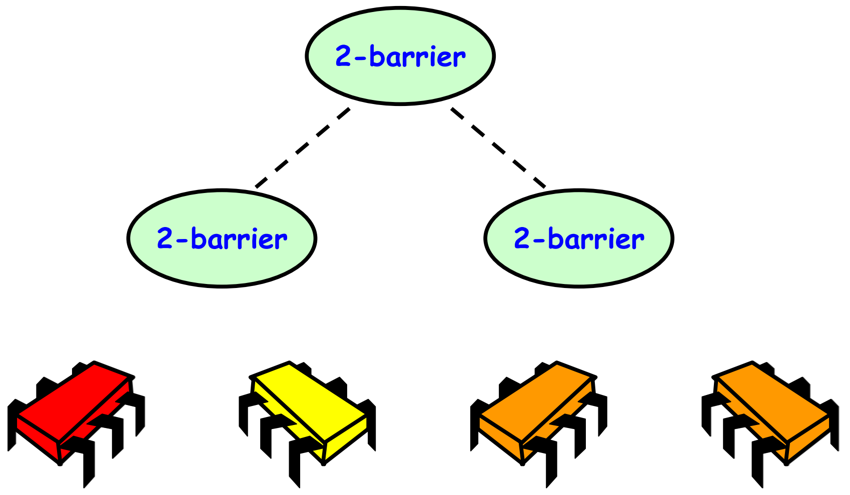


	Max spins	Elapsed time (seconds)	Improvement
Always-block	N/A	10529.5	0.0%
Always-spin	N/A	8256.3	21.5%
Fixed-spin	100	9108.0	13.5%
	200	8000.0	24.0%
Opt-known	1008	7881.4	25.1%
Opt-approx	1008	8171.2	22.3%
3-samples	1008	8011.6	23.9%
Random-walk	1008	7929.7	24.7%

Table 3: Elapsed times of Regsim using different spinning strategies.

From A. Karlin, K. Li, M. Manasse, and S. Owicki, "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor," *Proceedings of the 13th ACM Symposium on Operating Systems Principle,* 1991.

Combining Tree Barriers



Summary

- Cache is the key to implement a multiprocessor
  - Cache coherence is the design center
- Snooping and directory protocols similar;
  - bus makes snooping easier because of broadcast (snooping => uniform memory access)
  - Directory has extra data structure to keep track of state of all cache blocks
- Distributing directory
  - scalable shared address multiprocessor
  - Cache coherent, Non uniform memory access
- Synchronization
  - Require hardware support: atomic instructions
  - Need to be careful when using synchronization primitives on large multiprocessors