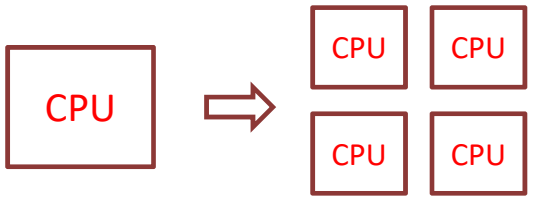
Multiprocessors and Thread-Level Parallelism

Slides modified from Prof. Jyh-Jiun Shann's Computer Architecture class

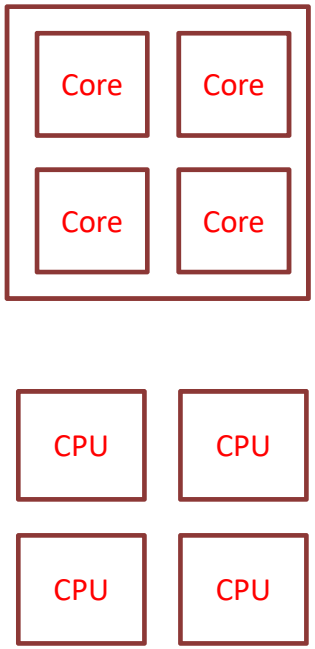
Introduction

- Focus is changed from high performance uniprocessor to Multiprocessor
  - It is harder to improve uniprocessor performance through ILP
  - Growing concern over power
  - It is easier to achieve higher performance by using commodity microprocessors than a single high performance microprocessor
- Other factors that reinforce this trend:
  - Growing interest in servers and server performance
  - Growth in data-intensive applications
  - Improved understanding of how to use multiprocessors effectively, especially where there is significant natural thread-level parallelism (TLP)
  - Easier to reuse a design investment rather than unique design



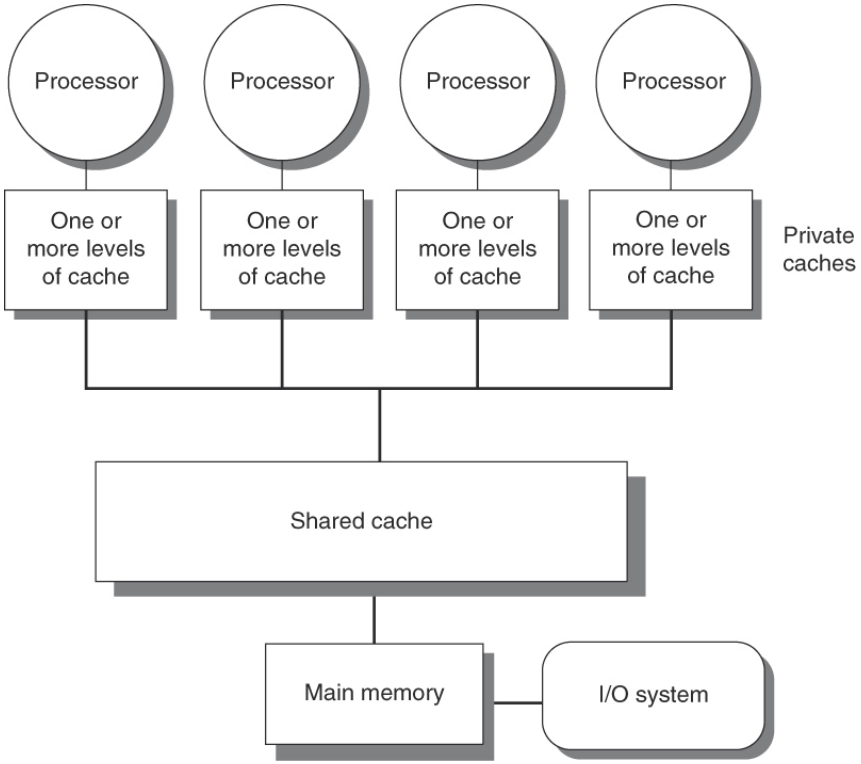
Multiprocessor design

- Multiprocessor: tightly coupled processors controlled by a single operating system and sharing memory address space
  - Multicore Multiprocessor: single-chip systems with multicore cores
  - Multichip Multiprocessor: system with separate chips, each chip can be single-core or multicore
  - Focus of this chapter
- Large-scale multiprocessor
  - Cluster: ultrascale computers built from very large number of processors, connected with networking technology
  - Warehouse-scale computers (WSC) : when there are tens of thousands of servers
  - Will not cover this (focus of next chapter)



Shared-memory multiprocessors

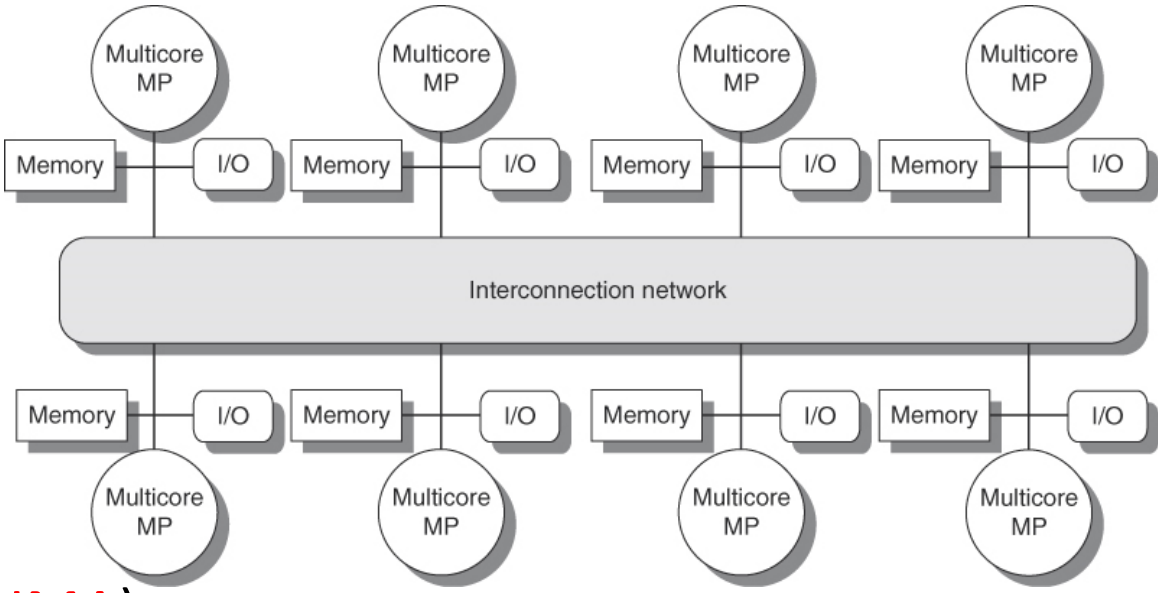
- Two types of shared-memory multiprocessor:
  - Centralized (shared-memory) multiprocessors
  - Distributed shared memory (DSM) multiprocessors
- Centralized multiprocessors
  - Share a single centralized memory all processors have equal access to
  - Small number of corers (normally < 100)
  - With large shared cache or shared memory to support memory bandwidth requirement
  - A.k.a. symmetric multiprocessors, uniform memory access multiprocessor (UMA)



Symmetric multiprocessor (UMA)

Distributed shared memory (DSM)

- Distributed memory to support larger processor counts and memory bandwidth
- Uses high-bandwidth interconnect such as
  - Directed networks (i.e., switches)
  - Indirect networks (meshes)



- A.k.a. nonuniform memory access (NUMA)
  - Access time depends on the location of a data word in memory
- Pros: cost-effective to scale the memory bandwidth if most accesses are local
- Cons: communicating b/t processors becomes more complex and needs higher latency

Challenges of Parallel Processing

- Challenges 1: Limited parallelism available in programs
  - Difficult to achieve high speedups
  - Need new algorithms
- Example: Assume computers run in two modes (1) Parallel mode with all processors fully used (2) Serial mode with only one processor in use. If we want to achieve a speedup of 80 with 100 processors, what fraction of the original computation can be sequential?

$$\text{Speedup} = \frac{1}{\frac{\text{Franction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Franction}_{\text{enhanced}})}$$

$$80 = \frac{1}{\frac{\text{Franction}_{\text{enhanced}}}{100} + (1 - \text{Franction}_{\text{enhanced}})}$$

$$\text{Franction}_{\text{enhanced}} = 0.9975$$

Only 0.25% of the programs can be sequential

Very difficult to achieve

Centralized Shared-Memory Architecture

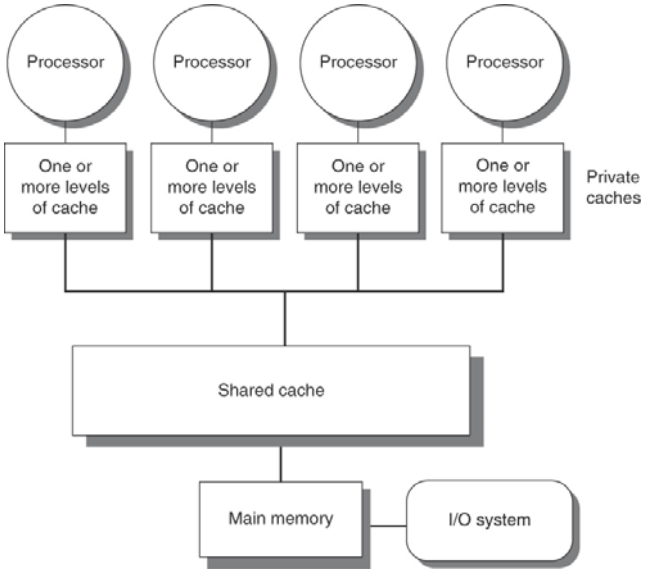
- Symmetric shared-memory machines support the caching of both shared and private data
  - Private data: data used by single processor
  - Shared data: data used by multiple processor
- Communication between processors is provided through reads and writes of the shared data
- When private data are cached, its location is migrated to cache, reducing average access time as well as memory bandwidth
  - Identical to uniprocessor
- When shared data are cached, data may be replicated in multiple caches
  - Reduce contention for shared data
  - Introduce new problem: **Cache Coherence**

Cache coherence problem

- A system is coherent if any read of a data item returns the most recently written value of that data item
- Cache coherence problem in shared-memory processors: different processors have different values for the same address

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

Basic Schemes for enforcing coherence



- Key idea
  - track the state of any sharing of a data block
  - Change state according to requests
- 2 schemes of cache coherence protocol
  - Snooping:
    - Cache are usually on a shared-memory bus, and are typically all accessible via some broadcast medium
    - Cache that has a copy of data block could track the block sharing status
    - all cache controllers monitor or snoop on the bus to determine if they have a copy of a block that is requested on the bus
    - Popular for symmetric shared-memory architecture
  - Directory based:
    - Sharing Status of blocks is kept in one location , called directory

Two styles of coherence protocol: write invalidate & write update

- Write invalidate protocol:
  - Writer send invalidation to all caches when data is modified (all other cache copies of the block are invalidated)
  - Ensure a processor has exclusive access to a data item before it writes that item
  - Used in most common protocols, both for snoopy and directory schemes
  - See example in the next page
- Write update
  - Also called write broadcast
  - Write propagate the update, i.e. update all the cache copies of a block when the block is modified
  - require more bandwidth => less popular now

Example: write invalidate

- Write invalidate example
- Assume: snooping, write-back is used

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Proc. A read X	Cache miss for X	0		0
Proc. B read X	Cache miss for X	0	0	0
Proc. A write 1 to X	Invalidation for X	1		0
Proc. B reads X	Cache miss for X	1	1	1

Become invalid

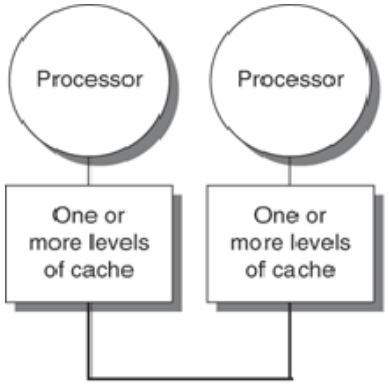
Example: write update

- Write update example
- Assume: snooping, write-back is used

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Proc. A read X	Cache miss for X	0		0
Proc. B read X	Cache miss for X	0	0	0
Proc. A write 1 to X	Write broadcast of X	1	1	1
Proc. B reads X	No Activity	1	1	1

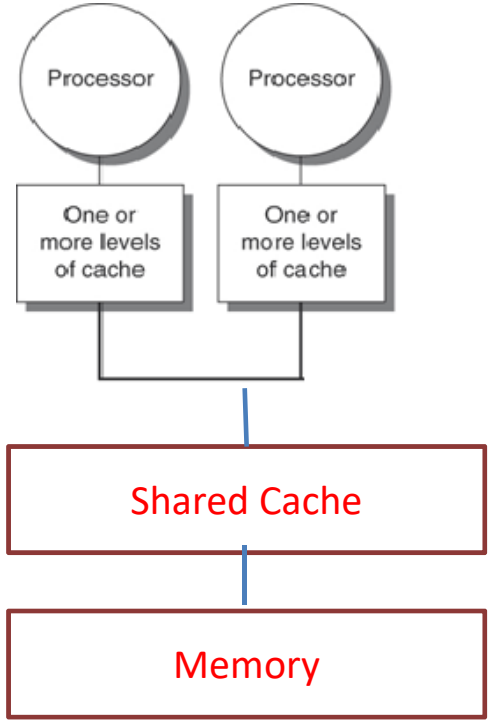
Basic Snooping Implementation Technique

- Implement snooping with write-invalidate on a symmetric shared-memory processor
- Use the bus to perform invalidates
  - Acquire bus access and broadcast address to be invalidated on bus
  - All processors continuously snoop on the bus, watching the address
  - Check if address on the bus is in their cache -> invalidate
- A write to a shared data item cannot complete until it obtains bus access
  - The first processor to obtain bus access will cause others' copies to be invalidated
- Assume atomic operations: No interrupt by other operations



How to locate data when a cache miss occurs?

- For a write-through cache: Memory
  - Memory always has the most updated data => From which the most recent value of a data item can be fetched
- For a write-back cache:
  - The most recent value of a data item can be in a cache
  - For cache that has a dirty copy of data (i.e. latest data), provide the data and abort memory access
    - Each processor snoops every address placed on the bus. If a processor finds it has a dirty copy of the block, it provides the cache block to the read request and abort the memory read access
  - More complex but generate lower memory traffic

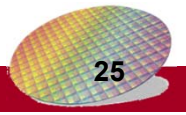


HW cache structure for implementing snooping

- Tag: used to identify the block
- Valid bit: 1 if block is valid. Makes invalidation easy to implement
- Dirty bit: used to identify if a block is modified
- Shared bit: shared mode or exclusive (unshared) mode
  - To track whether a block is shared or not => decide whether a write must generate invalidates
  - When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and marks the block as private (unshared). If another processor later requests this cache block, the state must be made shared again.

Scenarios in Snooping with write-Invalidate protocol

- Read hit (valid): reads the data block and continues
- Read miss : does not hold the block or has invalid block
  - Transfer the block from the shared memory (if write-through or if write-back & clean) or from the copy-holder (if write-back & dirty)
  - Set the corresponding valid-bit and shared-mode bit
    - The sole holder ? Yes -> change the shared-mode bit to exclusive
    - If the block before the read is exclusive? Yes -> set the shared-mode bit to shared



Scenarios in Snooping with write-Invalidate protocol

- Write hit:
  - Write hit to an exclusive cache block: (block owner)
    - Proceeds and continues
  - Write hit to a shared-read-only block
    - Need to obtain permission
    - Invalidate all cache copies
    - Completion of invalidation => write data and set the exclusive bit
    - The processor becomes the sole owner of the cache block until other read accesses arrive from other processor
      - Can be detected by snooping, then the block changes to the shared state
- Write miss: action similar to that of a write hit, except
  - A block copy is transferred to the processor and invalidate other copies

An example snooping protocol

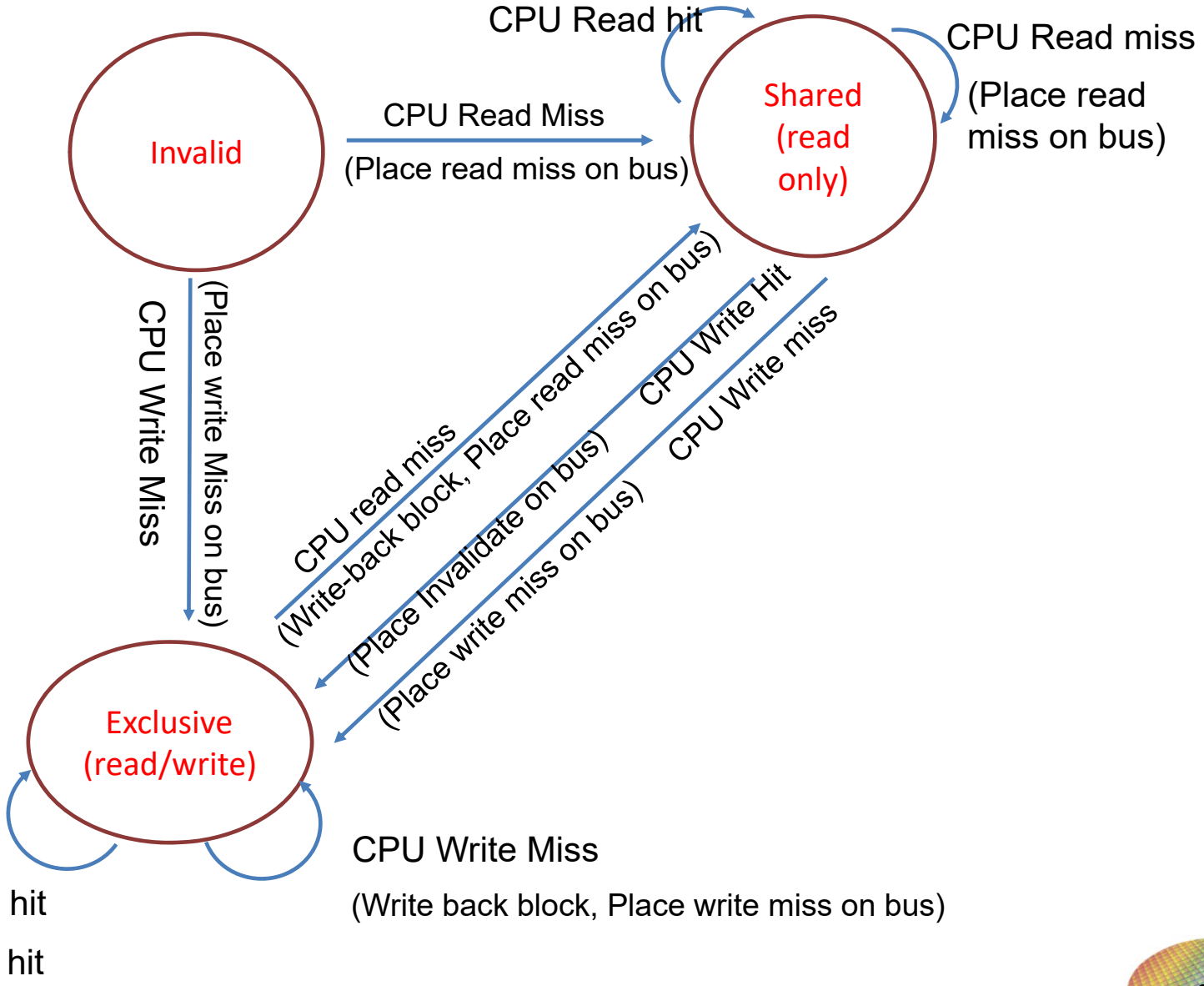
- Cache block states:
  - Invalid
  - Shared: the block is potentially shared
  - Modified (exclusion): the block has been updated in the cache=> the block is exclusive
- A cache coherence mechanism example for write-back cache
  - Receives requests from both the processors and the bus
  - Response to the request based on the type of request, whether it hits or misses in the cache, and the state of the cache block specified in the request => state diagram

State transition induced by the local processor

Cache block states:
- Invalid
- Shared
- Exclusive

Request from CPU
- CPU read hit
- CPU read miss
- CPU write hit
- CPU write miss

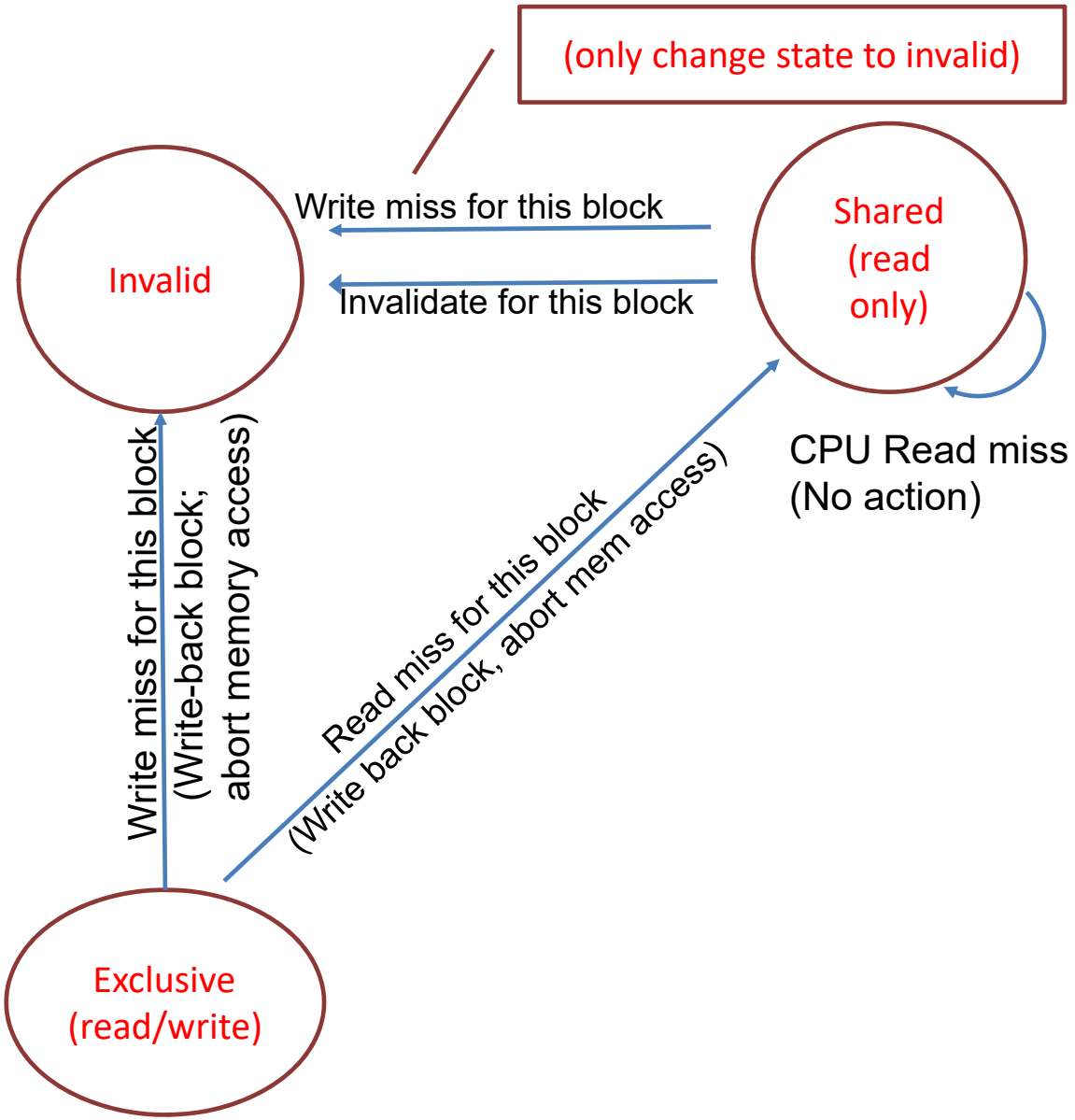
Request placed on bus
- Read miss for this block
- Write miss for this block
- Invalidate for this block



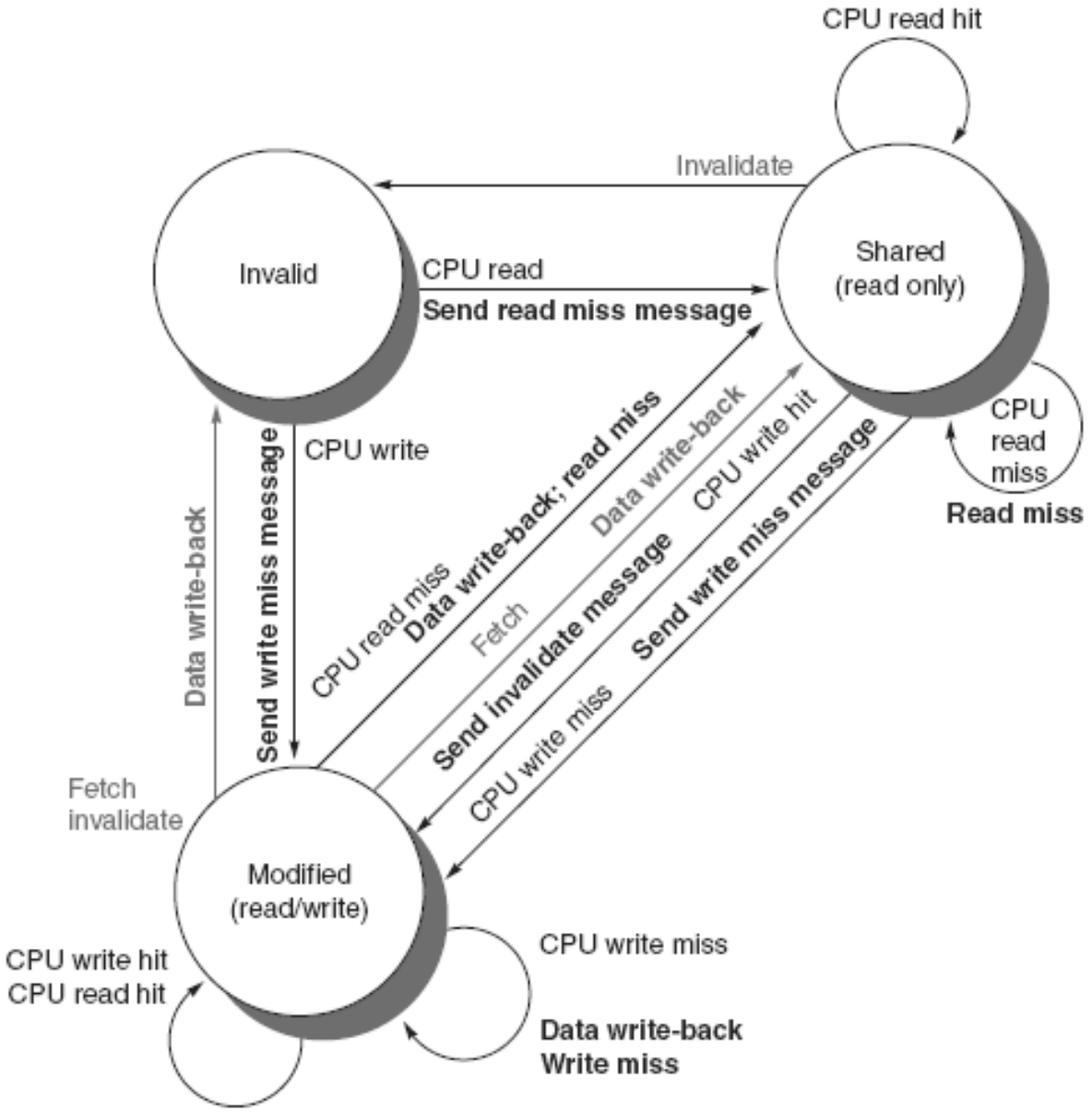
Request	Source	cache block State	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache
Read miss	Processor	Invalid	Normal miss	Place read miss on bus
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus
Read miss	Processor	Modified	Replacement	Address conflict miss; write back block, then place read miss on bus
Write hit	Processor	Modified	Normal hit	Write data in local cache
Write hit	Processor	Shared	Coherence	Place invalidate on bus; do not fetch data but only change the state
Write miss	Processor	Invalid	Normal miss	Place write miss on bus
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared
Invalidate	Bus	Shared	Coherence	Attempt to write shared block: invalidate the block
Write miss	Bus	Shared	Coherence	Attempt to write shared block: invalidate the block
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

State transition induced by the bus

Request from bus
- Read miss
- Write miss
- Invalidate



Final State Diagram



Backup Slides