# Cache Concepts and Performance

## DSC 315: Computer Organization & Operating Systems

**Dr. Laltu Sardar**

School of Data Science,
Indian Institute of Science Education and Research Thiruvananthapuram (IISER TVM)


February 17, 2026

# Memory hierarchy

1. Input-output and I/O System Design

# Cache Write Policies

- Cache improves performance by hiding memory latency
- Write policy controls how updates propagate to main memory
- Tradeoff between performance, complexity, and correctness

**Write-through**

- Writes update cache and main memory simultaneously
- Memory always holds the latest value
- Simple and predictable behavior

# Write-through: Pros and Cons

**Advantages**

- Strong consistency
- Simple hardware design
- Easier crash recovery

**Disadvantages**

- Higher write latency
- Increased memory traffic

# Write-back Cache

**Behavior**

- Writes update only the cache
- Memory updated later on cache eviction
- Modified lines marked as dirty

**Characteristics**

- Faster writes
- Memory may be temporarily stale

# Memory Stall

- A memory stall occurs when the CPU must wait for data or instructions
- Caused by cache misses, TLB misses, or memory ordering delays
- Pipeline stages become idle while waiting for memory
- Main memory latency can be hundreds of CPU cycles
- Memory stalls reduce IPC and overall performance

**Intuition:** the CPU pauses execution because the required data has not arrived yet.

# Write-back: Pros and Cons

**Advantages**

- Low write latency
- Reduced memory bandwidth usage
- High overall performance

**Disadvantages**

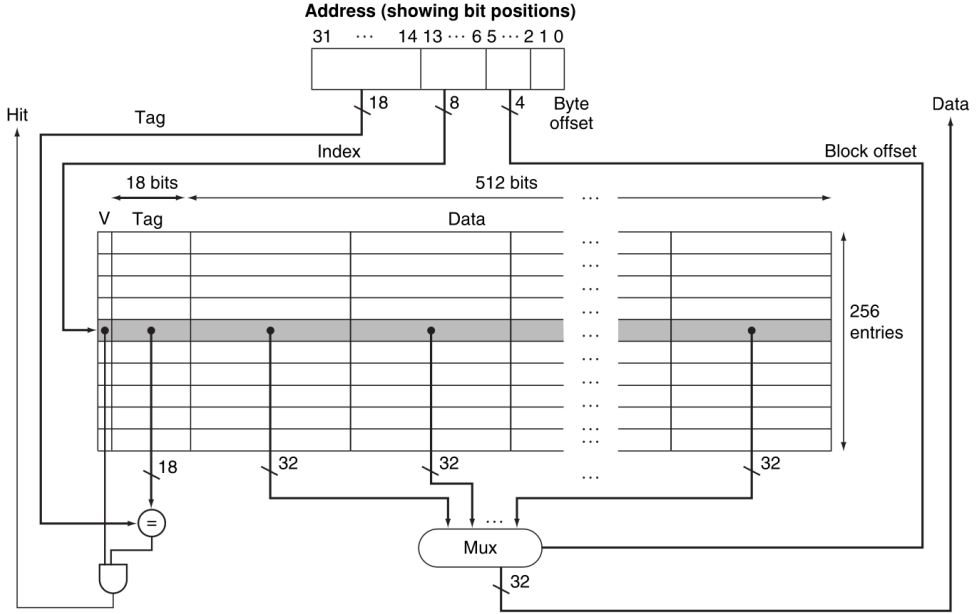
- More complex hardware
- Risk of data loss on power failure
- Harder cache coherence in multicore systems

# Summary and Use Cases

| Aspect | Write-through | Write-back |
|---|---|---|
| Write latency | High | Low |
| Memory consistency | Always current | Can be stale |
| Hardware complexity | Low | High |
| Crash safety | Better | Worse |
| Performance | Lower | Higher |

**Typical use**

- Write-through: embedded and safety-critical systems
- Write-back: modern CPUs and high-performance systems


Address (showing bit positions)
31 ··· 14 13 ··· 6 5 ··· 2 1 0
18
8
4
Byte offset
Hit
Tag
Index
Data
Block offset
18 bits
512 bits
V
Tag
Data
256 entries
18
32
32
32
=
Mux
32

**One-way set associative (direct mapped)**

| Block | Tag | Data |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |

**Two-way set associative**

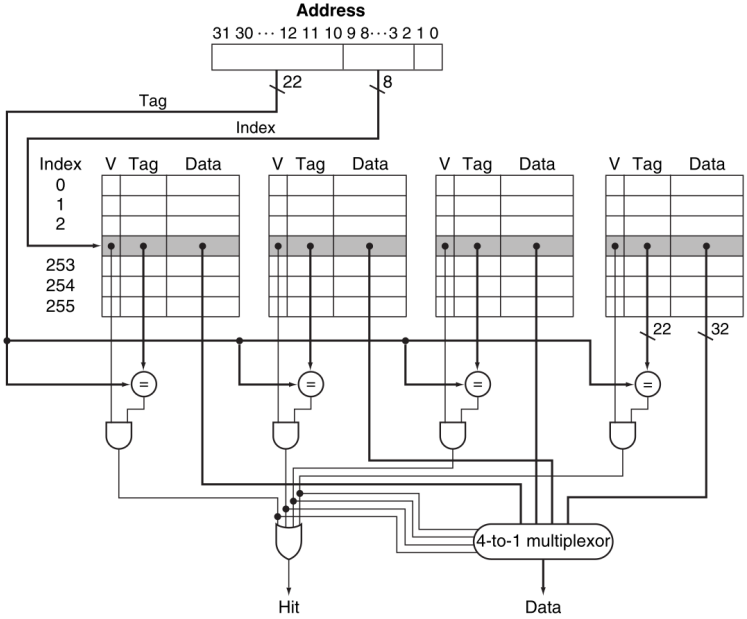
| Set | Tag | Data | Tag | Data |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

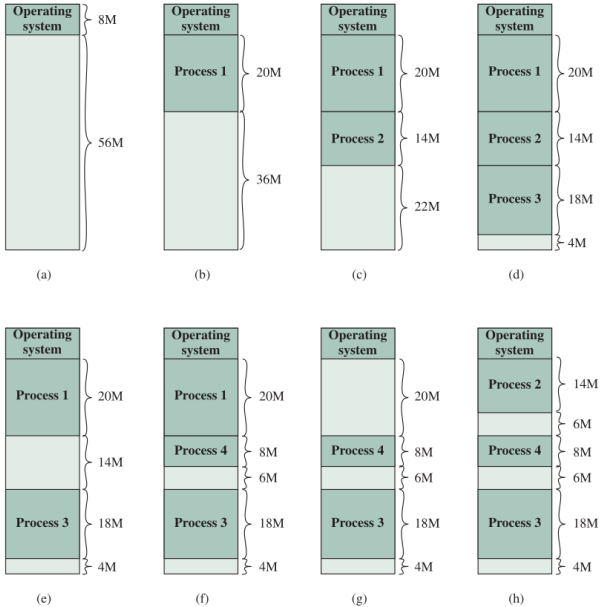
**Four-way set associative**

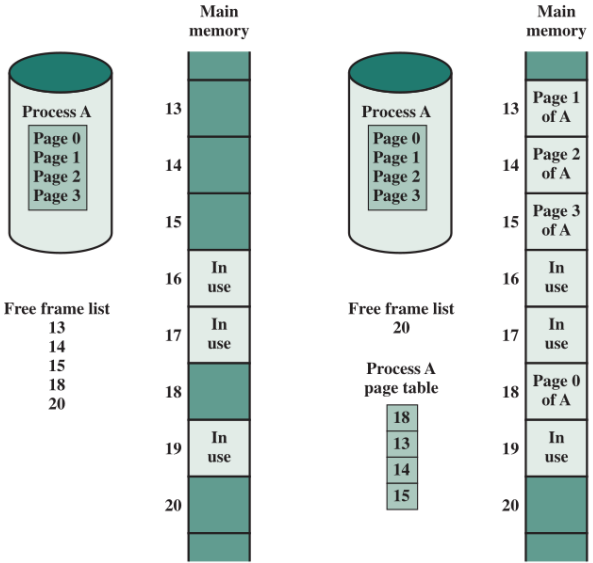
| Set | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | |
| 1 | | | | | | | | |

**Eight-way set associative (fully associative)**

| Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | |


Address
31 30 ··· 12 11 10 9 8···3 2 1 0
22
8
Tag
Index
Index
V
Tag
Data
0
1
2
253
254
255
22
32
4-to-1 multiplexor
Hit
Data


Operating system
8M
56M
(a)
Operating system
Process 1
20M
36M
(b)
Operating system
Process 1
20M
Process 2
14M
22M
(c)
Operating system
Process 1
20M
Process 2
14M
Process 3
18M
4M
(d)
Operating system
Process 1
20M
14M
Process 3
18M
4M
(e)
Operating system
Process 1
20M
Process 4
8M
6M
Process 3
18M
4M
(f)
Operating system
20M
Process 4
8M
6M
Process 3
18M
4M
(g)
Operating system
Process 2
14M
6M
Process 4
8M
6M
Process 3
18M
4M
(h)


Main memory
Process A
Page 0
Page 1
Page 2
Page 3
13
14
15
16
17
18
19
20
In use
In use
In use
Free frame list
13
14
15
18
20
(a) Before
Main memory
Process A
Page 0
Page 1
Page 2
Page 3
Page 1 of A
Page 2 of A
Page 3 of A
In use
In use
Page 0 of A
In use
Free frame list
20
Process A page table
18
13
14
15
(b) After

# Why Cache Performance Matters

- Modern CPUs execute instructions much faster than main memory
- Cache misses cause the CPU to stall
- Overall performance depends heavily on how often and how long the CPU waits for memory

# CPU Execution Time

CPU execution time can be expressed as

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

Total clock cycles include:

- CPU execution cycles
- Memory stall cycles

# Memory-Stall Clock Cycles

**Definition**

Memory-stall clock cycles are cycles during which the CPU is stalled waiting for data or instructions from memory.

$$\text{Memory Stall Cycles} = \text{Read-Stall Cycles} + \text{Write-Stall Cycles}$$

# Read-Stall Cycles

Read-stall cycles occur due to cache misses on read operations.

**Sources of Read Stalls**

- Instruction cache misses
- Data cache misses on loads

$$\text{Read-Stall Cycles} = \text{Read Misses} \times \text{Miss Penalty}$$

# Write-Stall Cycles

Write-stall cycles depend on the cache write policy.

**Possible Causes**

- Write-through cache with a write buffer that becomes full
- Write misses that require fetching a block from memory

$$\text{Write-Stall Cycles} = \text{Write Misses} \times \text{Miss Penalty}$$

# Instruction Miss Cycles

Instruction miss cycles occur when the instruction cache misses.

$$\text{Instruction Miss Cycles} = \text{Instruction Miss Rate} \times \text{Instructions} \times \text{Miss Penalty}$$

These stalls directly delay instruction fetch and pipeline progress.

# Data Miss Cycles

Data miss cycles are caused by load and store instructions missing in the data cache.

$$\text{Data Miss Cycles} = \text{Data Miss Rate} \times \text{Memory Accesses} \times \text{Miss Penalty}$$

They often dominate performance in data-intensive programs.

# Putting It All Together

**Total Memory Stall Cycles**

$$\text{Memory Stall Cycles} = \text{Instruction Miss Cycles} + \text{Data Miss Cycles}$$

**Total CPU Clock Cycles**

$$\text{CPU Clock Cycles} = \text{Execution Cycles} + \text{Memory Stall Cycles}$$

# Average Memory Access Time

**Definition**

Average Memory Access Time (AMAT) measures the average time to access memory, including hits and misses.

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

# Improving Cache Performance

- Reduce miss rate
  - Larger cache
  - Better associativity
  - Improved block placement
- Reduce miss penalty
  - Multilevel caches
  - Critical-word-first and early restart
- Reduce hit time
  - Simple cache structures
  - Pipelined caches

# Why Cache Misses Occur

- Cache misses increase memory stall cycles
- Reducing miss rate directly improves CPU performance
- Cache organization plays a major role in miss behavior

# Fully Associative Cache

**Key Idea**

Any memory block can be placed in any cache line.

- No index field in the address
- Entire address tag is compared with all cache tags
- Lowest conflict miss rate
- Requires expensive hardware for parallel tag comparison

# Set-Associative Cache

**Key Idea**

The cache is divided into sets, each containing multiple blocks.

- A block maps to exactly one set
- It can be placed in any line within that set
- Compromise between direct-mapped and fully associative caches

# Locating a Block in the Cache

**Address Breakdown**

$$\text{Address} = \text{Tag} \mid \text{Set Index} \mid \text{Block Offset}$$

- Set index selects the set
- Tag comparison determines hit or miss
- Block offset selects the requested word or byte

# Choosing Which Block to Replace

When a cache miss occurs and the set is full, one block must be replaced.

**Replacement Policies**

- Least Recently Used (LRU)
- First-In First-Out (FIFO)
- Random replacement

LRU generally gives the lowest miss rate but is more complex to implement.

# Size of Tags vs Set Associativity

- Increasing associativity reduces conflict misses
- Higher associativity increases:
  - Tag storage per cache line
  - Hardware complexity for tag comparison
- Fully associative caches require the largest tag fields

**Trade-off**

Lower miss rate versus higher cost and longer hit time.

# Reducing Miss Penalty with Multilevel Caches

**Key Idea**

Use multiple levels of cache with increasing size and access time.

- L1 cache: small, fast, low hit time
- L2 cache: larger, slower, lower miss rate
- L3 cache: even larger, shared among cores

# Effect of Multilevel Caches

**Effective Miss Penalty**

$$\text{Effective Miss Penalty} = \text{L1 Miss Rate} \times \text{L2 Miss Penalty}$$

Each additional cache level significantly reduces the average memory access time.

# Finally

- Associativity reduces conflict misses
- Replacement policies affect miss rate
- Higher associativity increases tag size and hardware cost
- Multilevel caches reduce effective miss penalty

# Virtual Memory

Section 5.4

# Motivation for Virtual Memory

- Programs may be larger than physical memory
- Multiprogramming requires memory isolation
- Efficient use of limited main memory is critical

**Key Idea**

Virtual memory gives each process the illusion of a large, private, contiguous memory space.

# Virtual Address Space

- Each process has its own virtual address space
- Virtual addresses are mapped to physical addresses
- Only active portions of a program are kept in memory

$$\text{Virtual Address} \longrightarrow \text{Physical Address}$$

# Paging

- Virtual memory is divided into fixed-size pages
- Physical memory is divided into page frames
- Page size is typically a power of 2

**Terminology**

- Page: unit of virtual memory
- Page frame: unit of physical memory

# Virtual Address Format

For a page size of $2^n$ bytes:

$$\text{Virtual Address} = \text{Page Number} \mid \text{Page Offset}$$

- Page offset uses $n$ bits
- Page number uses remaining bits

# Page Table

- Page table maps virtual page numbers to physical frames
- Each process has its own page table

## Page Table Entry

Typically contains:

- Frame number
- Valid or invalid bit
- Protection bits
- Dirty bit

# Address Translation Using Page Table

1. Extract page number from virtual address
2. Use page number to index page table
3. Obtain physical frame number
4. Concatenate with page offset

$$\text{Physical Address} = \text{Frame Number} \mid \text{Page Offset}$$

# Page Fault

**Definition**

A page fault occurs when a referenced page is not in physical memory.

- Operating system is invoked
- Required page is fetched from disk
- Page table is updated
- Instruction is restarted

# Demand Paging

- Pages are loaded only when needed
- Reduces memory usage
- Increases efficiency for large programs

**Key Property**

Programs exhibit locality of reference.

# Locality of Reference

- Temporal locality: recently used pages are reused
- Spatial locality: nearby addresses are accessed

Locality makes demand paging effective in practice.

# Translation Lookaside Buffer (TLB)

- A small, fast cache for page table entries
- Stores recent virtual to physical translations

**Goal**

Reduce address translation time.

# Effective Access Time with TLB

Let:

- $t_{TLB}$ = TLB access time
- $t_M$ = main memory access time
- $h$ = TLB hit ratio

$$\text{EAT} = h \times (t_{TLB} + t_M) + (1 - h) \times (t_{TLB} + 2t_M)$$

# Page Replacement

When memory is full, a page must be replaced.

## Common Algorithms

- Least Recently Used (LRU)
- First-In First-Out (FIFO)
- Optimal replacement

# Page Replacement Goal

- Minimize page fault rate
- Reduce disk I O operations
- Improve overall system performance

# Effective Access Time with Page Faults

Let:

- $p$ = page fault rate
- $t_{PF}$ = page fault service time

$$\text{EAT} = (1 - p) \times t_M + p \times t_{PF}$$

# Thrashing

**Definition**

Thrashing occurs when the system spends most of its time handling page faults.

- Insufficient physical memory
- Excessive multiprogramming
- Poor page replacement decisions

# Summary

- Virtual memory separates address space from physical memory
- Paging enables efficient memory usage
- TLB improves address translation speed
- Page faults and replacement dominate performance

# A Common Framework for Memory Hierarchies

Section 5.5

# Parallelism and Private Caches

- Modern processors exploit parallelism using multiple cores
- Each core typically has private L1 instruction and data caches
- Private caches reduce latency and increase bandwidth
- However, multiple cached copies of the same memory block may exist

**Key Challenge**

Maintaining a consistent view of shared memory across all processors.

# Shared Memory Multiprocessor Model

- All processors share a single physical address space
- Each processor executes its own instruction stream
- Memory can be accessed by any processor
- Caches replicate memory blocks for performance

Without coordination, replicated data can become inconsistent.

# Definition of Cache Coherence

**Cache Coherence**

A system is cache coherent if it ensures that all processors see a consistent value for any shared memory location.

- A read returns the most recent write
- Writes are eventually visible to all processors
- All processors observe writes to the same location in the same order

# The Cache Coherence Problem

- Processor P1 reads a memory block into its cache
- Processor P2 also reads the same block
- P1 modifies the block locally
- P2 still holds an outdated copy

**Consequence**

P2 may read stale data unless coherence is enforced.

# Coherence Requirements

A cache coherence protocol must satisfy:

- **Write propagation**: a write by one processor eventually becomes visible to all
- **Write serialization**: writes to the same location are seen in the same order by all processors
- **Read correctness**: a read returns the value of the most recent write

# Basic Approaches to Cache Coherence

- Snooping based protocols
- Directory based protocols

**Key Difference**

How coherence information is communicated among caches.

# Snooping Based Coherence

- All caches are connected via a shared bus
- Each cache controller monitors all bus transactions
- Cache controllers take action when they observe relevant addresses
- Common in small scale multiprocessors

Broadcast based communication simplifies design but limits scalability.

# Snooping Transactions

- Read miss generates a bus read request
- Write generates invalidation or update messages
- Other caches snoop the request and respond accordingly

Bus traffic grows rapidly as processor count increases.

# Write Invalidate Protocol

- On a write, all other cached copies are invalidated
- Only one cache holds a writable copy
- Subsequent reads by other processors cause cache misses

**Advantage**

Lower bandwidth usage compared to write update.

# Write Update Protocol

- On a write, updated data is broadcast to all caches
- All cached copies remain valid
- Readers always see up to date data

**Disadvantage**

High bus bandwidth consumption for frequent writes.

# MESI Cache Coherence Protocol

MESI is a widely used write invalidate snooping protocol.

## Cache Line States

- Modified
- Exclusive
- Shared
- Invalid

# MESI State: Modified

- Cache line contains modified data
- Memory copy is stale
- This cache has exclusive ownership
- Data must be written back on eviction

# MESI State: Exclusive and Shared

- Exclusive: block is clean and present in only one cache
- Shared: block is clean and may exist in multiple caches
- Reads are allowed in both states
- Writes require state transition

# MESI State: Invalid

- Cache line does not contain valid data
- Any access results in a cache miss
- Line must be refetched from memory or another cache

# Directory Based Coherence

- Used in large scale multiprocessors
- Each memory block has an associated directory entry
- Directory tracks which caches hold the block

Avoids broadcast traffic and improves scalability.

# Directory Entry Structure

A directory entry typically contains:

- Block state
- Owner cache identifier
- Sharer list or bit vector

Directory based protocols trade storage for scalability.

# False Sharing

- Occurs when processors modify different variables
- Variables reside in the same cache block
- Causes unnecessary invalidations

False sharing severely degrades parallel performance.

# Reducing False Sharing

- Pad data structures to cache block boundaries
- Separate frequently written variables
- Improve memory layout for parallel programs

# Performance Cost of Coherence

- Increased cache miss rate due to invalidations
- Additional memory latency from coherence traffic
- Reduced scalability with increasing core count

Efficient coherence is critical for parallel speedup.

# Summary

- Cache coherence ensures correctness in shared memory systems
- Snooping and directory protocols enforce coherence
- MESI is a practical and widely deployed protocol
- False sharing is a major performance pitfall

# Parallelism and Memory Hierarchies: Cache Coherence

Section 5.8

# Parallelism and the Need for Cache Coherence

- Modern processors improve performance by executing multiple instruction streams in parallel
- Each processor core typically has private L1 caches to reduce memory access latency
- Caches replicate data blocks from main memory to improve locality and bandwidth
- Parallel programs frequently share data across cores

**Fundamental Issue**

Multiple cached copies of the same memory location can lead to inconsistent views of memory.

# Shared Memory Multiprocessor Architecture

- All processors share a single physical address space
- Any processor can read or write any memory location
- Each processor has private caches holding copies of memory blocks
- Main memory acts as the backing store

Without a coordination mechanism, updates made by one processor may not be visible to others.

# What Is Cache Coherence?

**Cache Coherence**

Cache coherence is the property that ensures all processors observe a consistent value for each shared memory location.

- A read must return the value of the most recent write
- Writes by one processor must eventually be seen by all others
- All processors must observe writes to the same address in the same order

# Illustration of the Coherence Problem

- Processor P1 reads variable X into its cache
- Processor P2 also reads X into its cache
- P1 updates X locally in its cache
- P2 continues to read the old value of X

**Result**

Program correctness is violated unless coherence is enforced.

# Coherence Requirements

Any cache coherence protocol must guarantee:

- **Write propagation**: updates eventually reach all caches
- **Write serialization**: all processors see writes in the same order
- **Read correctness**: a read returns the value of the most recent write

These properties are necessary for correct shared memory semantics.

# Coherence vs Consistency

- Cache coherence concerns a single memory location
- Memory consistency concerns ordering of accesses to different locations
- A system can be coherent but still exhibit surprising behaviors without a strong consistency model

Cache coherence is a prerequisite for memory consistency.

# Basic Approaches to Cache Coherence

- Snooping based coherence protocols
- Directory based coherence protocols

**Key Distinction**

How caches communicate coherence information to one another.

# Snooping Based Cache Coherence

- All caches are connected to a shared communication medium
- Every cache controller monitors all memory transactions
- Requests are broadcast to all caches
- Caches take action based on observed requests

Snooping protocols are simple but do not scale well.

# Snooping Protocol Operations

- Read miss generates a bus read request
- Write generates a bus write or invalidate request
- Other caches snoop the bus and update their state

Bus contention increases rapidly with processor count.

# Write Invalidate Protocol

- On a write, all other cached copies are invalidated
- Only one cache holds a writable copy
- Subsequent reads by other processors cause cache misses

**Benefit**

Lower coherence traffic for multiple writes to the same block.

# Write Update Protocol

- On a write, updated data is broadcast to all caches
- All cached copies remain valid and consistent
- Readers immediately see updated values

**Drawback**

High bandwidth consumption for write intensive workloads.

# MESI Cache Coherence Protocol

- A write invalidate snooping protocol
- Widely implemented in commercial processors
- Each cache line has one of four states

**States**

Modified, Exclusive, Shared, Invalid

# MESI State: Modified

- Cache line contains modified data
- Memory copy is stale
- Only one cache holds the block
- Write back required on eviction

# MESI State: Exclusive

- Cache line is clean
- Present in exactly one cache
- Memory copy is up to date
- Write can proceed without invalidation

# MESI State: Shared

- Cache line may be present in multiple caches
- All copies are clean
- Write requires invalidation of other copies

# MESI State: Invalid

- Cache line does not contain valid data
- Any access results in a cache miss
- Block must be fetched from memory or another cache

# Directory Based Cache Coherence

- Designed for large scale multiprocessors
- Avoids broadcast communication
- A directory keeps track of block ownership and sharers

Scales better than snooping based approaches.

# Directory Entry Information

Each directory entry typically stores:

- State of the memory block
- Identity of the owner cache
- List or bitmap of sharer caches

Directory storage overhead grows with system size.

# False Sharing

- Occurs when different processors update different variables
- Variables reside in the same cache block
- Causes repeated invalidations and cache misses

False sharing degrades performance without true data sharing.

# Reducing False Sharing

- Align shared variables to cache block boundaries
- Pad frequently written variables
- Redesign data structures for parallel access

# Performance Impact of Cache Coherence

- Increased memory access latency
- Higher cache miss rate due to invalidations
- Reduced scalability for write shared data

Efficient coherence protocols are critical for parallel speedup.

# Summary

- Cache coherence maintains correctness in shared memory systems
- Snooping and directory protocols enforce coherence
- MESI is a practical and widely used protocol
- False sharing is a major performance bottleneck


**Dr. Laltu Sardar**, Assistant Professor, IISER Thiruvananthapuram
laltu.sardar@iisertvm.ac.in, laltu.sardar.crypto@gmail.com
Course webpage: https://laltu-sardar.github.io/courses/corgos_2026.html.


Carl Hamacher, Zvonko Vranesic, and Safwat Zaky.
*Computer Organization*.
McGraw-Hill, 6th edition, 2012.


John L. Hennessy and David A. Patterson.
*Computer Architecture: A Quantitative Approach*.
Morgan Kaufmann, 6th edition, 2017.


Vincent P. Heuring and Harry F. Jordan.
*Computer System Design and Architecture*.
Pearson, 2nd edition, 2007.


David A. Patterson and John L. Hennessy.
*Computer Organization and Design: The Hardware/Software Interface*.
Morgan Kaufmann, 4th edition, 2014.


William Stallings.
*Computer Organization and Architecture: Designing for Performance*.
Pearson, 10th edition, 2016.