Cache Memory

COE 501

Computer Architecture

Prof. Muhamed Mudawar

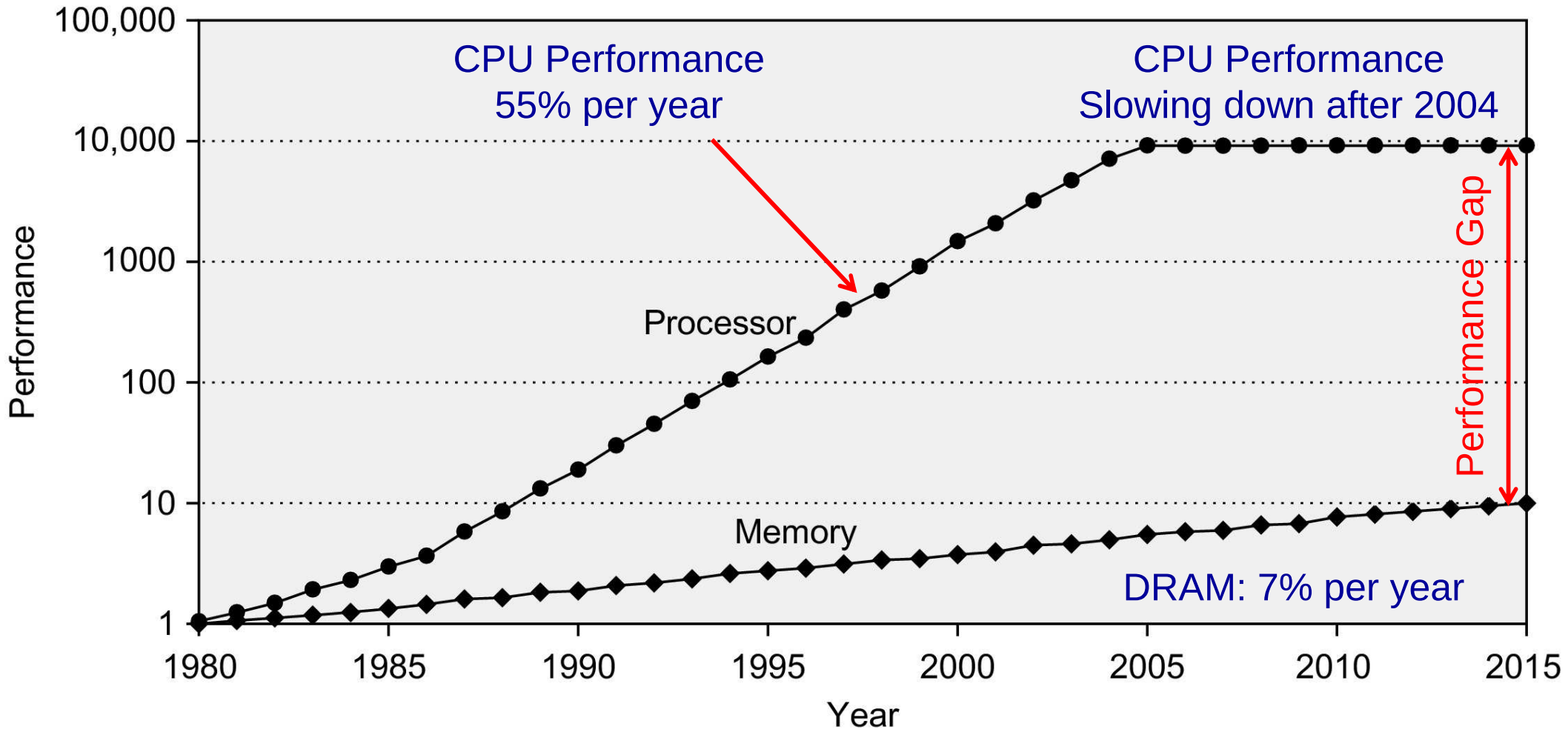
Computer Engineering Department

King Fahd University of Petroleum and Minerals

The Need for Cache Memory

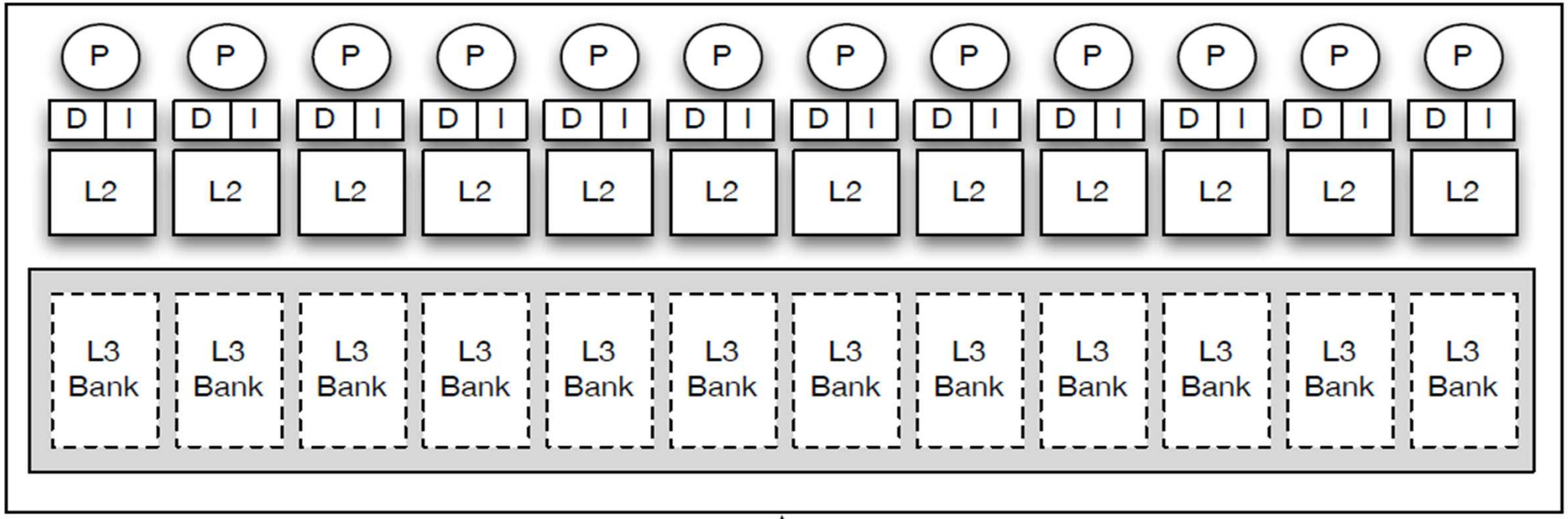
- Widening speed gap between CPU and main memory
  - Processor operation takes less than 0.5 ns
  - Off-chip main memory typically requires 50 to 100 ns to access
- Each instruction involves at least one memory access
  - One memory access to fetch the instruction
  - A second memory access for load and store instructions
- Memory bandwidth limits the instruction execution rate
- Cache memory can help bridge the CPU-memory gap
- Cache memory is small in size but fast

Processor-Memory Performance Gap



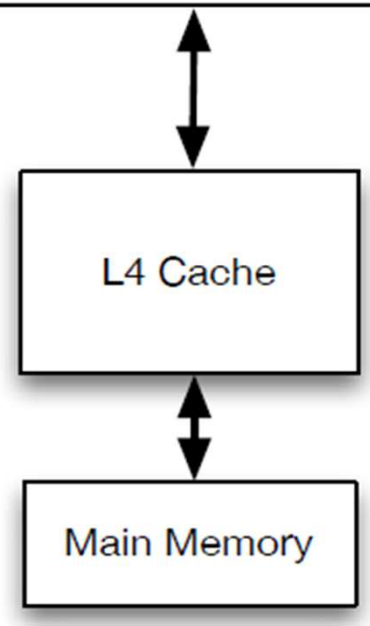
- 1980 – No caches in microprocessor
- 1995 – Two-level caches in microprocessor

Typical Memory Hierarchy



L1 Cache: 32 – 64KB (SRAM)
Instruction / Data Cache
1 ns (2-3 cycles) access time

L2 Cache: 256KB – 1MB (SRAM)
3-10 ns (6-20 cycles) access time



L3 Cache: 8MB – 96MB (eDRAM)
10-20 ns access time

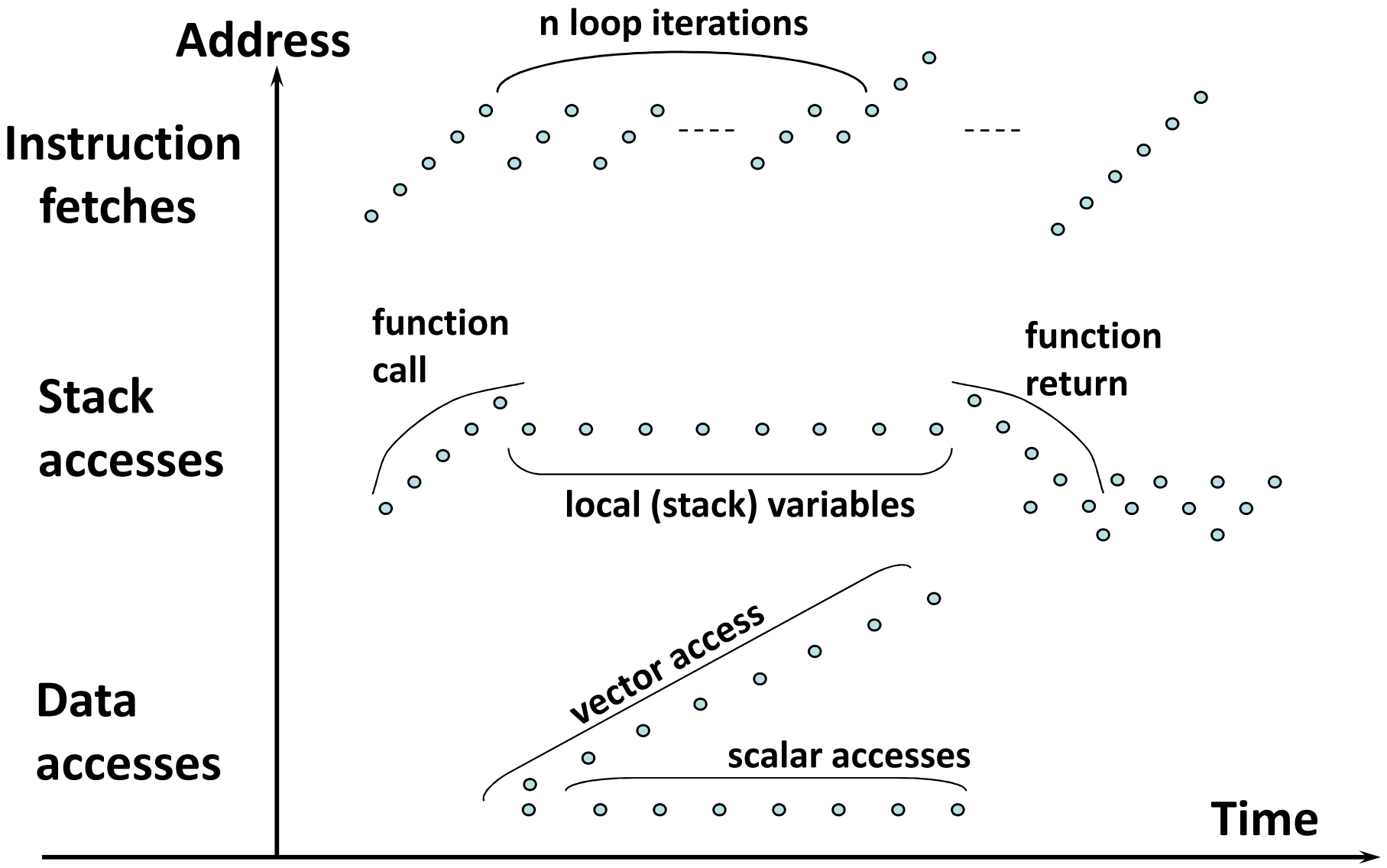
Off-chip L4 Cache (larger than L3)
20-40 ns access time

8GB – 256GB Main Memory
DRAM, 50-100 ns access time

Principle of Locality of Reference

- Programs access small portion of their address space
  - At any time, only a small set of instructions & data is needed
- Temporal Locality (locality in time)
  - If a memory location is accessed, it is likely that it will be accessed again
  - Same loop instructions are fetched each iteration
  - Same procedure may be called and executed many times
- Spatial Locality (locality in space)
  - Tendency to access contiguous instructions/data in memory
  - Sequential execution of Instructions
  - Traversing arrays sequentially

Memory Reference Patterns



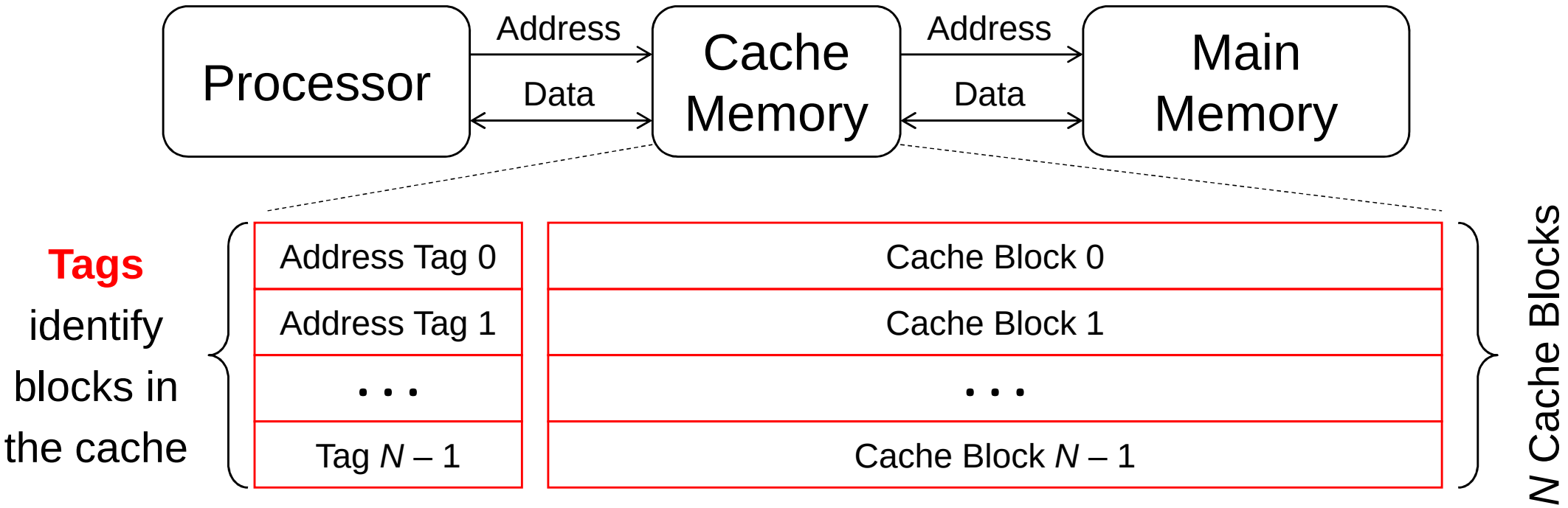
What is a Cache Memory ?

- Small and fast (SRAM) memory technology
  - Stores a subset of instructions & data currently being accessed
- Used to reduce the average memory access time
- Caches exploit temporal locality by …
  - Keeping recently accessed data closer to the processor
- Caches exploit spatial locality by …
  - Moving blocks consisting of multiple contiguous words
- Goal is to achieve
  - Fast speed of cache memory access
  - Balance the cost of the memory system

Four Basic Questions on Caches

- Q1: Where can a block be placed in a cache?
  - Block placement
  - Direct Mapped, Set Associative, Fully Associative
- Q2: How is a block found in a cache?
  - Block identification
  - Block address, tag, index
- Q3: Which block should be replaced on a miss?
  - Block replacement
  - Random, FIFO, LRU, NRU
- Q4: What happens on a write?
  - Write strategy
  - Write Back or Write Through (Write Buffer)

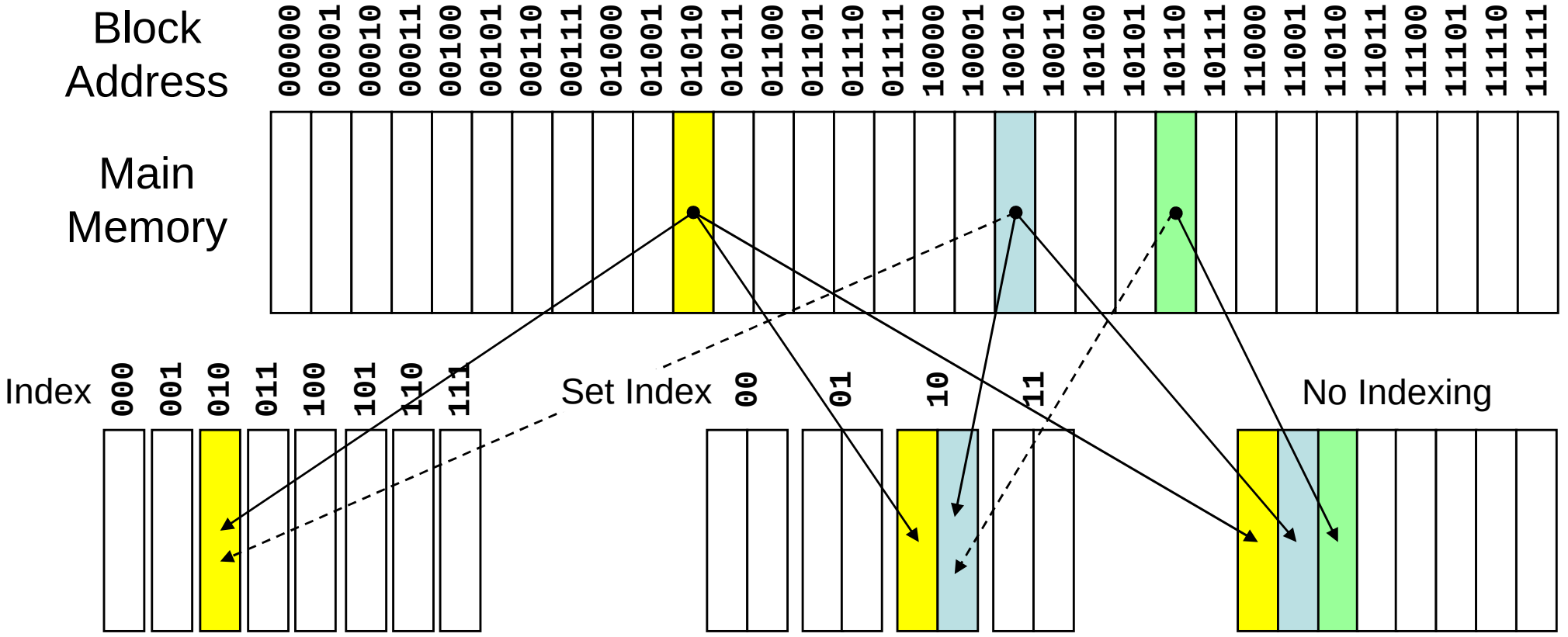
Inside a Cache Memory



❖ **Cache Block** (or Cache Line)

- ✧ Unit of data transfer between main memory and a cache
- ✧ Large block size ➔ Less tag overhead + Burst transfer from DRAM
- ✧ Cache block size = 64 bytes in recent caches

Block Placement



Direct Mapped Cache

A memory block is mapped to one cache index

Cache index = lower 3 bits of block address

2-way Set Associative

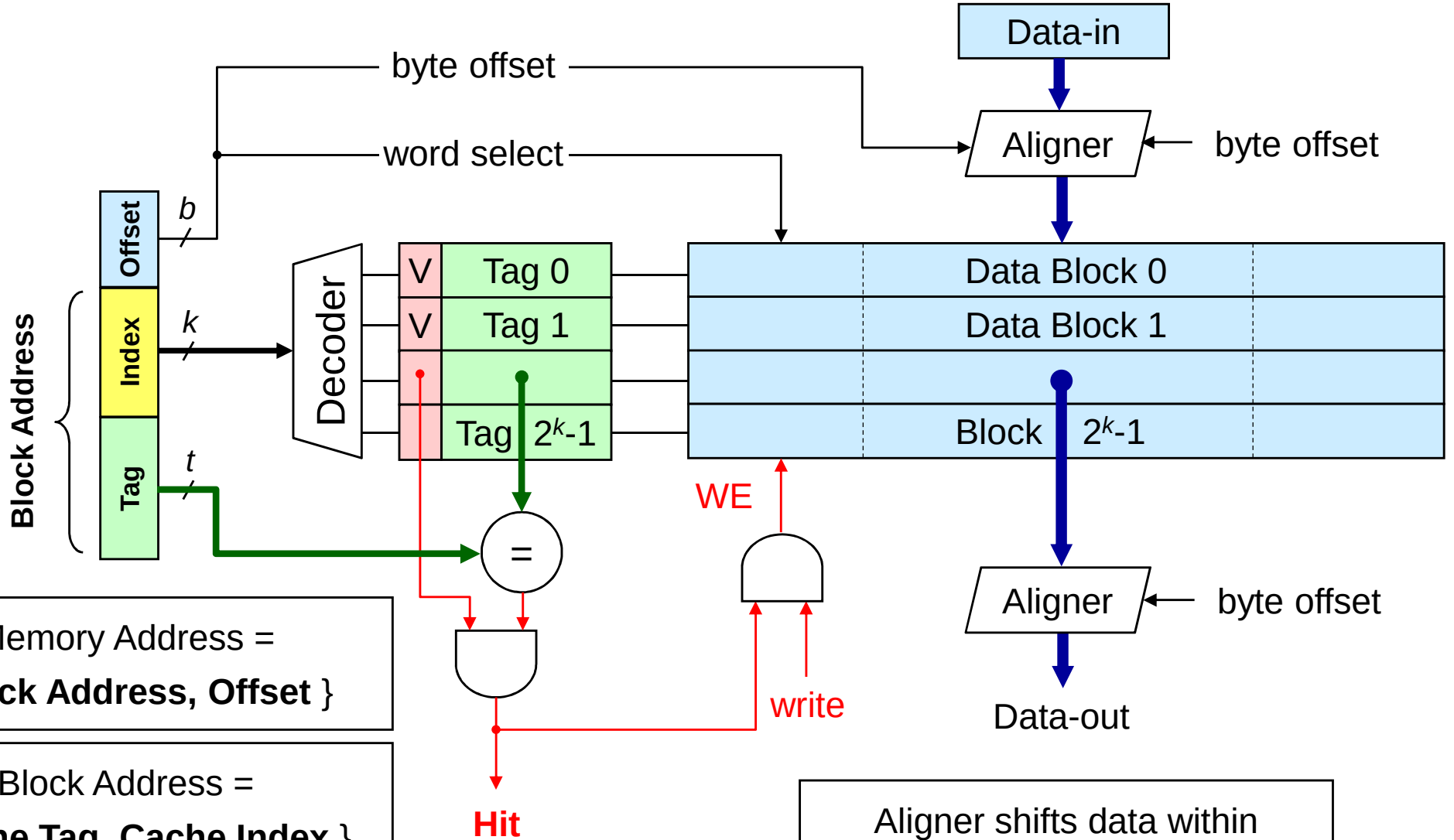
A memory block placed anywhere inside a set

Set index = lower 2 bits of block address

Fully Associative

A memory block can be placed anywhere inside the cache

Direct Mapped Cache



Memory Address = { **Block Address, Offset** }

Block Address = { **Cache Tag, Cache Index** }

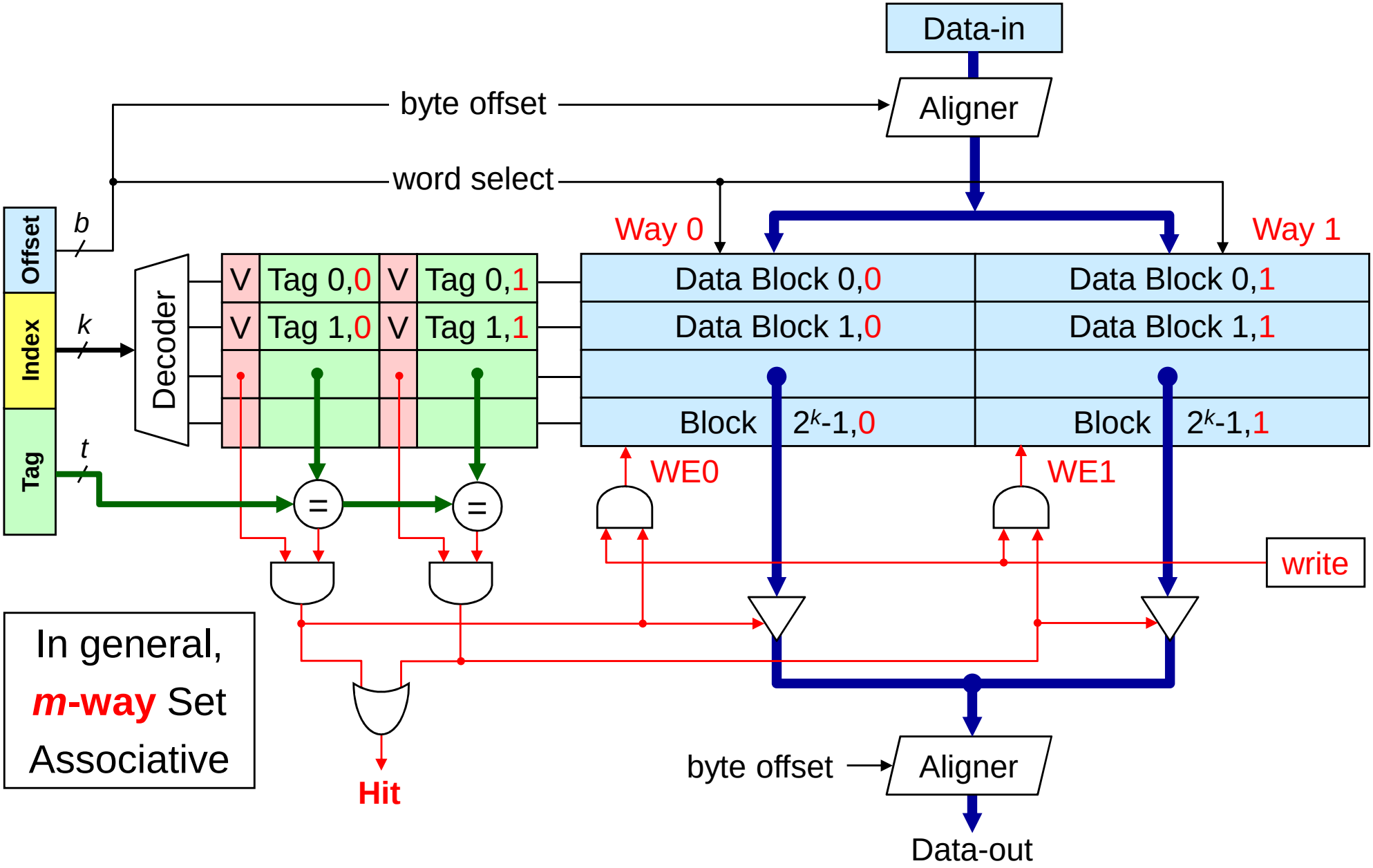
Offset = { **Word, Byte offset** }

Aligner shifts data within word according to byte offset

Direct Mapped Cache

- **Block address:** identifies block in memory
- **Offset:** used to access word and byte within a block
- **Index:** used for direct cache access, decoded to select one entry
- **Tag:** used for block identification
  - Compare address tag against cache entry tag
  - If the tags are equal and the entry is **valid** then cache **Hit**; otherwise **Miss**
- Cache index field is *k*-bit long ➔ Number of cache blocks is 2^k
- Block Offset field is *b*-bit long = { word select, byte offset }
  - Number of bytes in a block is 2^b
- Tag field is *t*-bit long ➔ Memory address = $t + k + b$ bits
- Cache data size = 2^{k+b} bytes

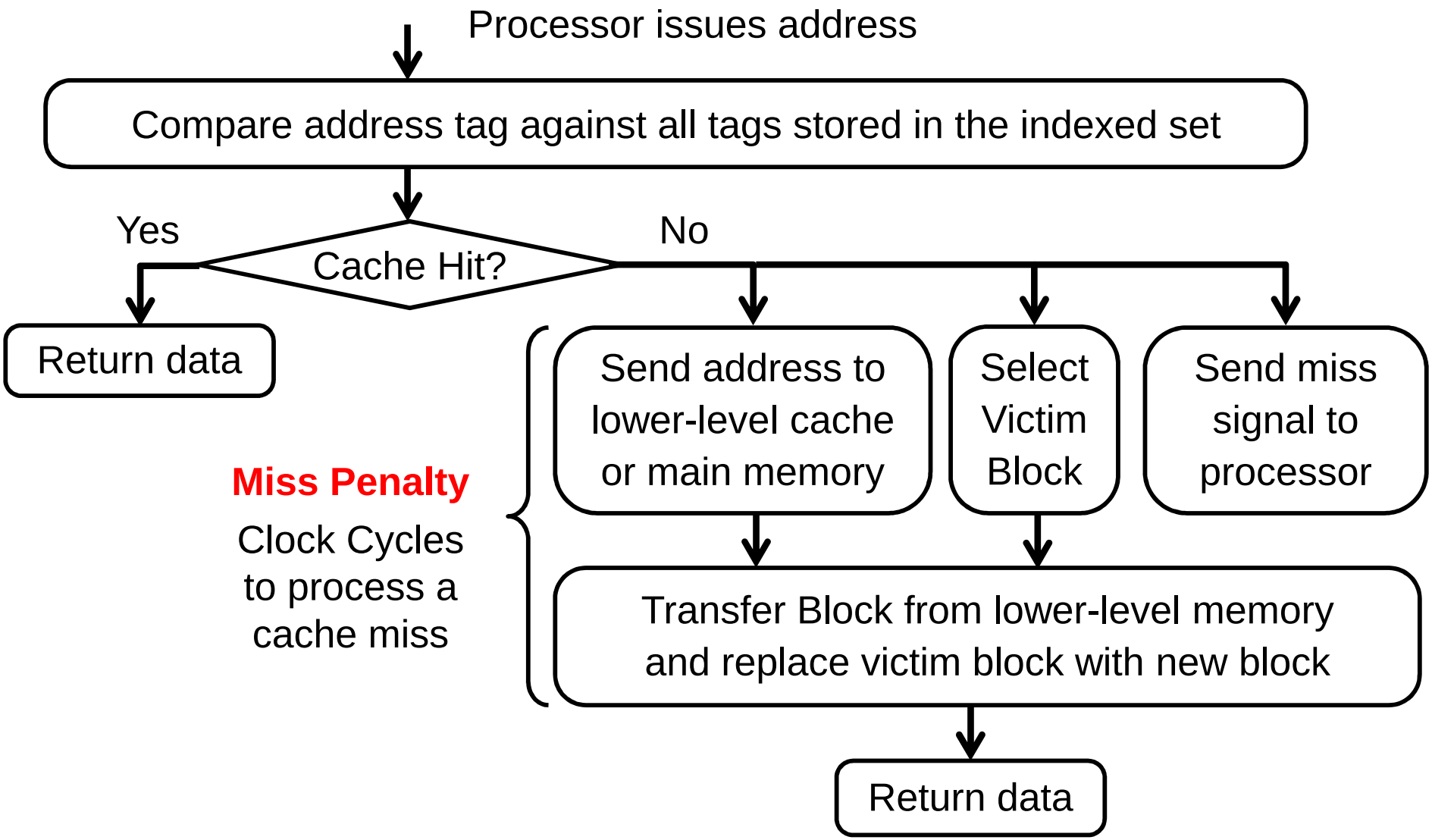
Set-Associative Cache



Set-Associative Cache

- A **set** is a group of blocks that can be indexed
- One set = ***m*** blocks ➔ ***m*-way** set associative
- Set index field is k-bit long ➔ Number of Sets = 2^k
- Set index is decoded and only one set is examined
  - ***m*** tags are checked in parallel using ***m*** comparators
  - If address tag matches a stored tag within set then **Cache Hit**
  - Otherwise: **Cache Miss**
- Cache data size = $m \times 2^{k+b}$ bytes (2^b bytes per block)
- A direct-mapped cache has one block per set (m = 1)
- A fully-associative cache has only one set (k = 0)

Handling a Cache Read



Replacement Policy

- Which block to replace on a cache miss?
- No choice for direct-mapped cache
- ***m*** choices for *m*-way set associative cache
- **Random replacement**
  - Candidate block is selected randomly
  - One counter for all sets (0 to *m* – 1): incremented on every cycle
  - On a cache miss, replace block specified by counter
- **First In First Out (FIFO) replacement**
  - Replace oldest block in set (Round-Robin)
  - One counter per set (0 to *m* – 1): specifies oldest block to replace
  - Counter is incremented on a cache miss

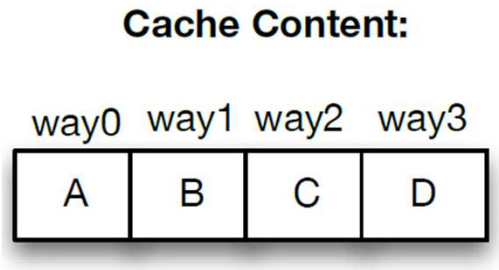
LRU Replacement Policy

❖ LRU: Replace Least Recently Used Block in a Set

- ✧ LRU state must be updated on every cache hit
- ✧ If *m* = 2, there are only 2 permutations ➔ a single LRU bit is needed
- ✧ If *m* = 4 then 4! = 24 permutations. If *m* = 8 then 8! = 40320 permutations
- ✧ Pure LRU is difficult to implement for large *m*

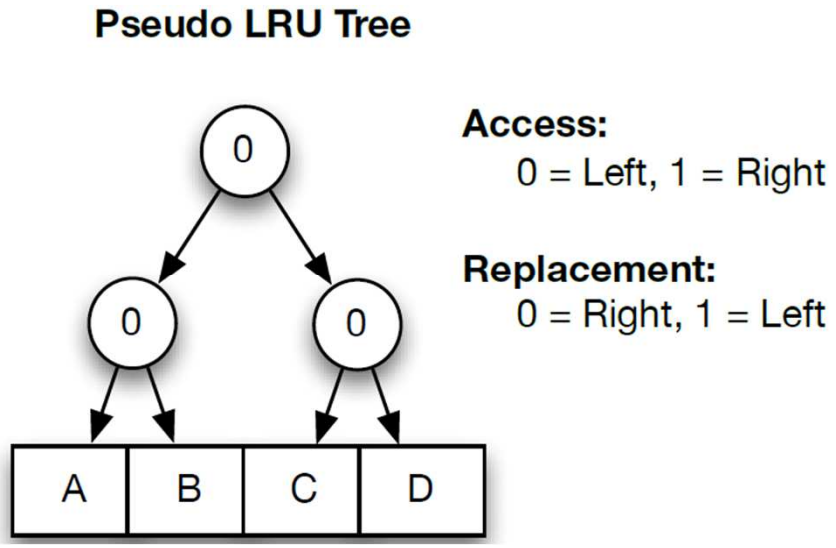
❖ Pseudo LRU Tree

- ✧ LRU approximation that requires (*m* – 1) LRU bits per set



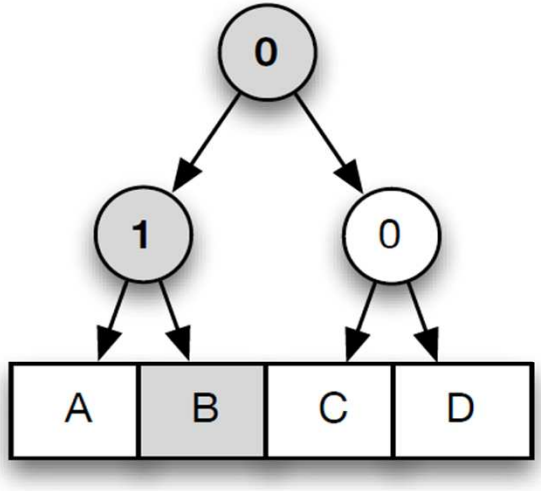
On cache hit, track the MRU block

On cache miss, replace non-MRU block

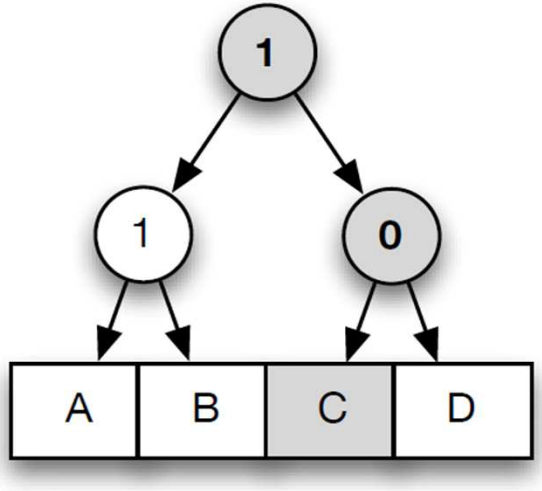


Access Stream: B, C, A, D, E

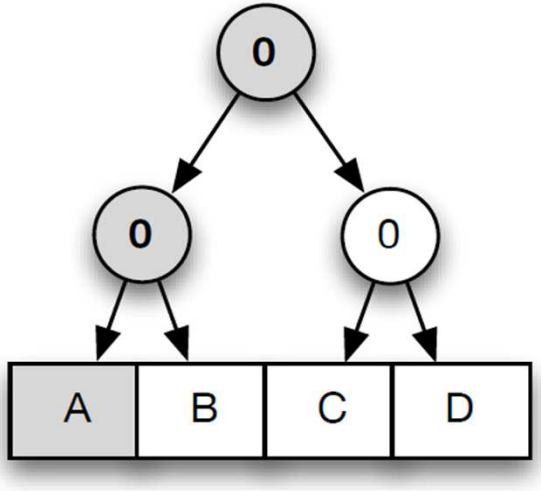
After access to B



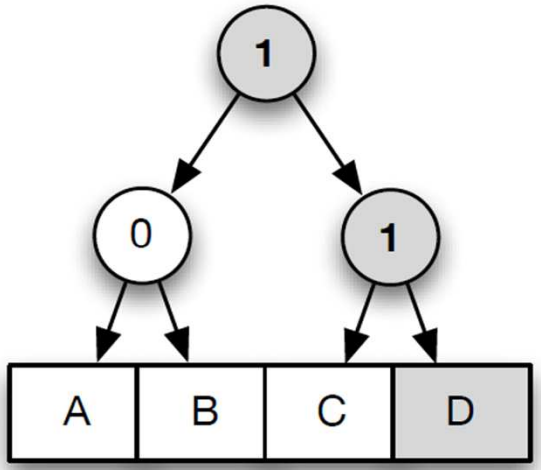
After access to C



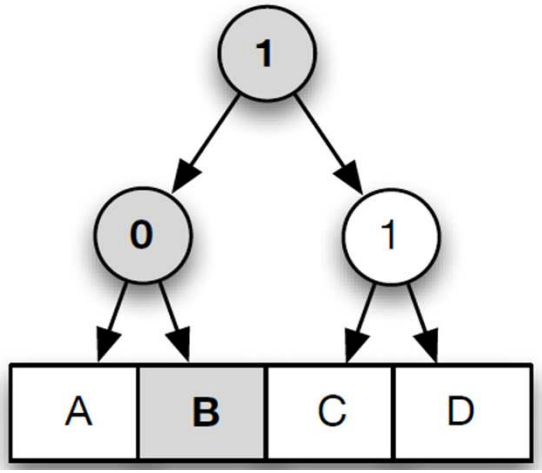
After access to A



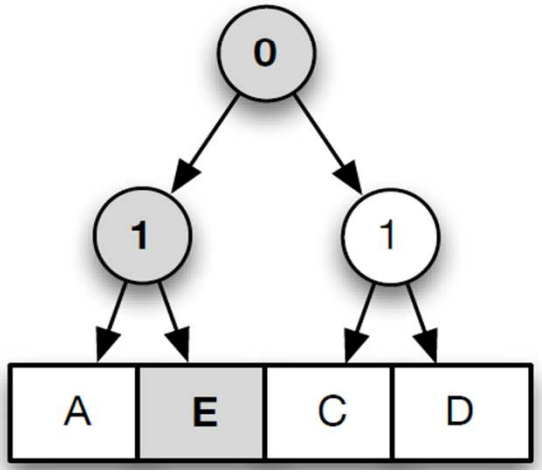
After access to D



Finding a block to replace
to make room for E
Block B is selected



After access to E,
E replaces B,
Bits flipped along the path



Comparing Random, FIFO, and LRU

- Data cache misses per 1000 instructions
  - 10 SPEC2000 benchmarks on Alpha processor
  - Block size = 64 bytes
  - LRU outperforms FIFO and Random for a small cache
  - Little difference between LRU and Random for a large cache
- Random is the simplest to implement: one counter for all sets
- Pseudo LRU Tree requires (m – 1) replacement bits per set

	2-way			4-way			8-way		
Size	LRU	Rand	FIFO	LRU	Rand	FIFO	LRU	Rand	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Write Policy

❖ Write Through

- ✧ Writes update data cache and lower-level (L2) cache
- ✧ Cache control bit: only a Valid bit is needed
- ✧ Lower-level cache always has latest data, which simplifies data coherency
- ✧ Can always discard cached data when a block is replaced

❖ Write Back

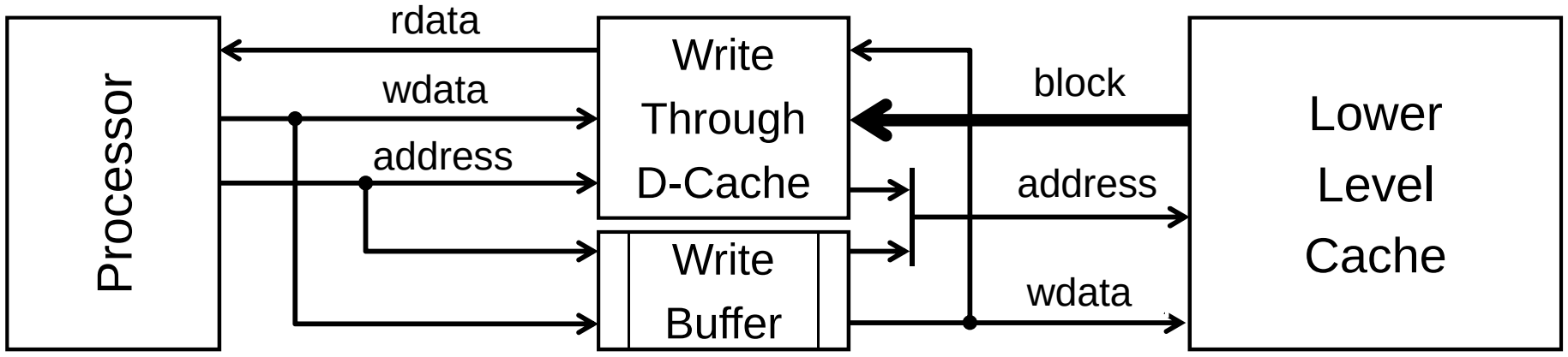
- ✧ Writes update data cache only
- ✧ Two cache control bits: Valid and Modified bits are required
- ✧ Modified bit indicates that cache block ≠ memory block
- ✧ Modified cached block is written back when replaced
- ✧ Multiple writes to same cache block require only one write-back
- ✧ Uses less memory bandwidth than write-through and less energy

Write Miss Policy

- What happens on a write miss?
- **Write Allocate**
  - Allocate new block in data cache
  - Write miss acts like a read miss, block is fetched then updated
- **Write No-Allocate**
  - Send data to lower-level (L2) cache
  - Data cache is not modified
- Either write-miss policy can be combined with either write policy
- **Write back** caches typically use **write allocate** on a write miss
  - Reasoning: subsequent writes will be captured in the cache
- **Write-through** caches might choose **write no-allocate**
  - Reasoning: writes must still go to lower level memory

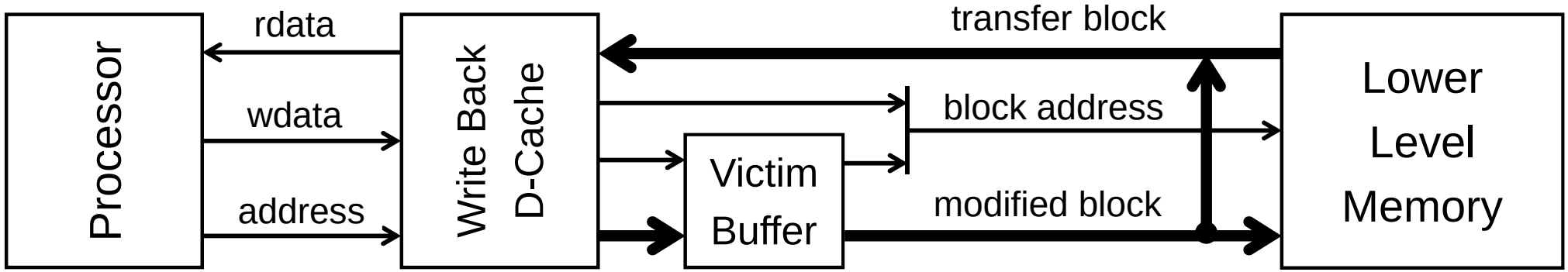
Write Buffer

- Write buffer is a queue that holds: address + write-data (wdata)
  - Write-through: all writes are sent to lower-level cache
- Buffer decouples the write from the memory bus writing
  - Write occurs **without stalling processor**, until buffer is full
- **Problem:** write buffer may hold data on a read miss
  - If address is found, return data value in write buffer
  - Transfer block from lower level cache and update D-cache



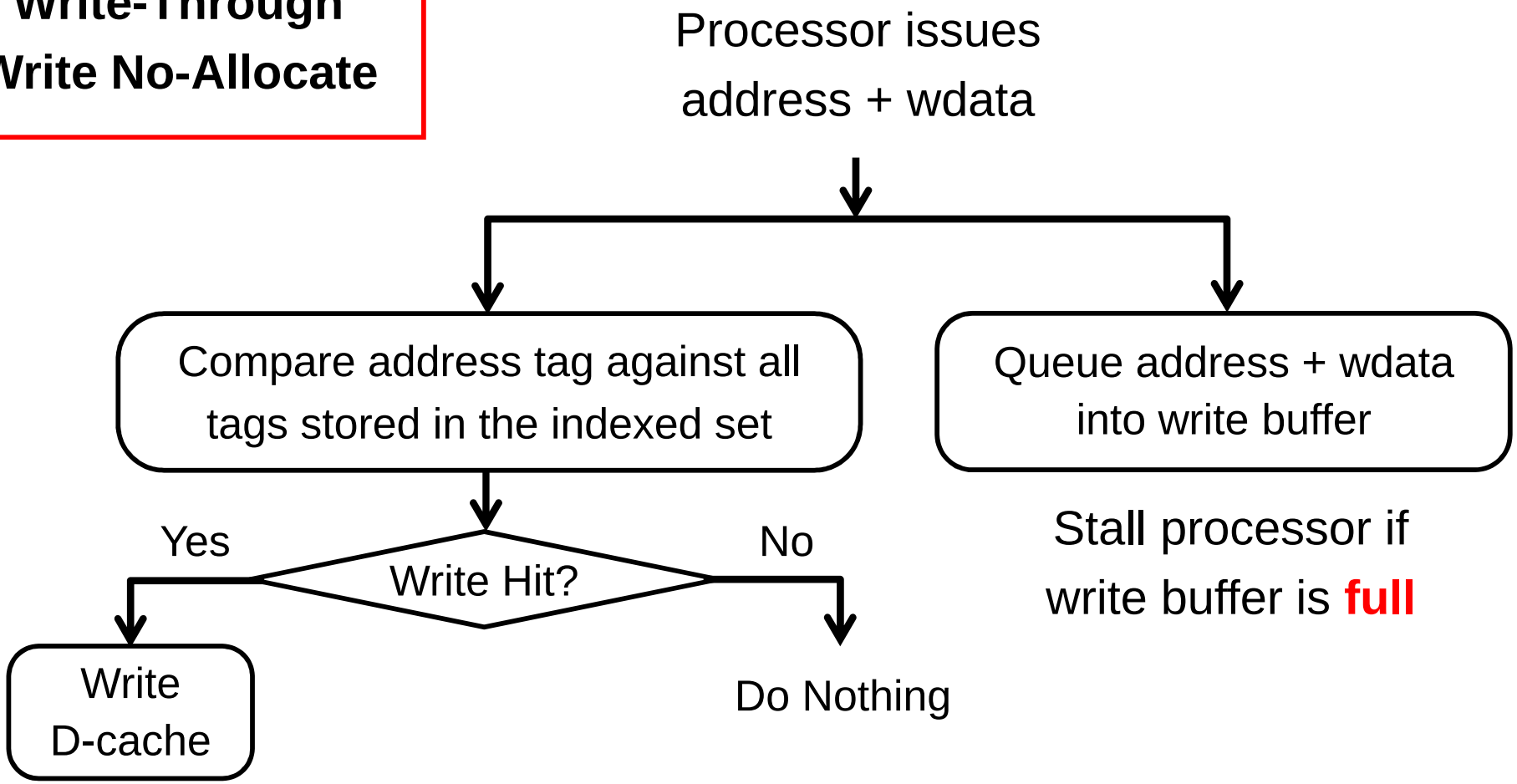
Victim Buffer

- Used by modified evicted blocks and their addresses
  - When a modified block is replaced (evicted) in the D-cache
  - Prepares a modified block to be written back to lower memory
- Victim buffer decouples the write back to lower memory
  - Giving priority to read miss over write back to reduce miss penalty
- Problem: Victim buffer may hold block on a cache miss
  - Solution: transfer modified block in victim buffer into data cache

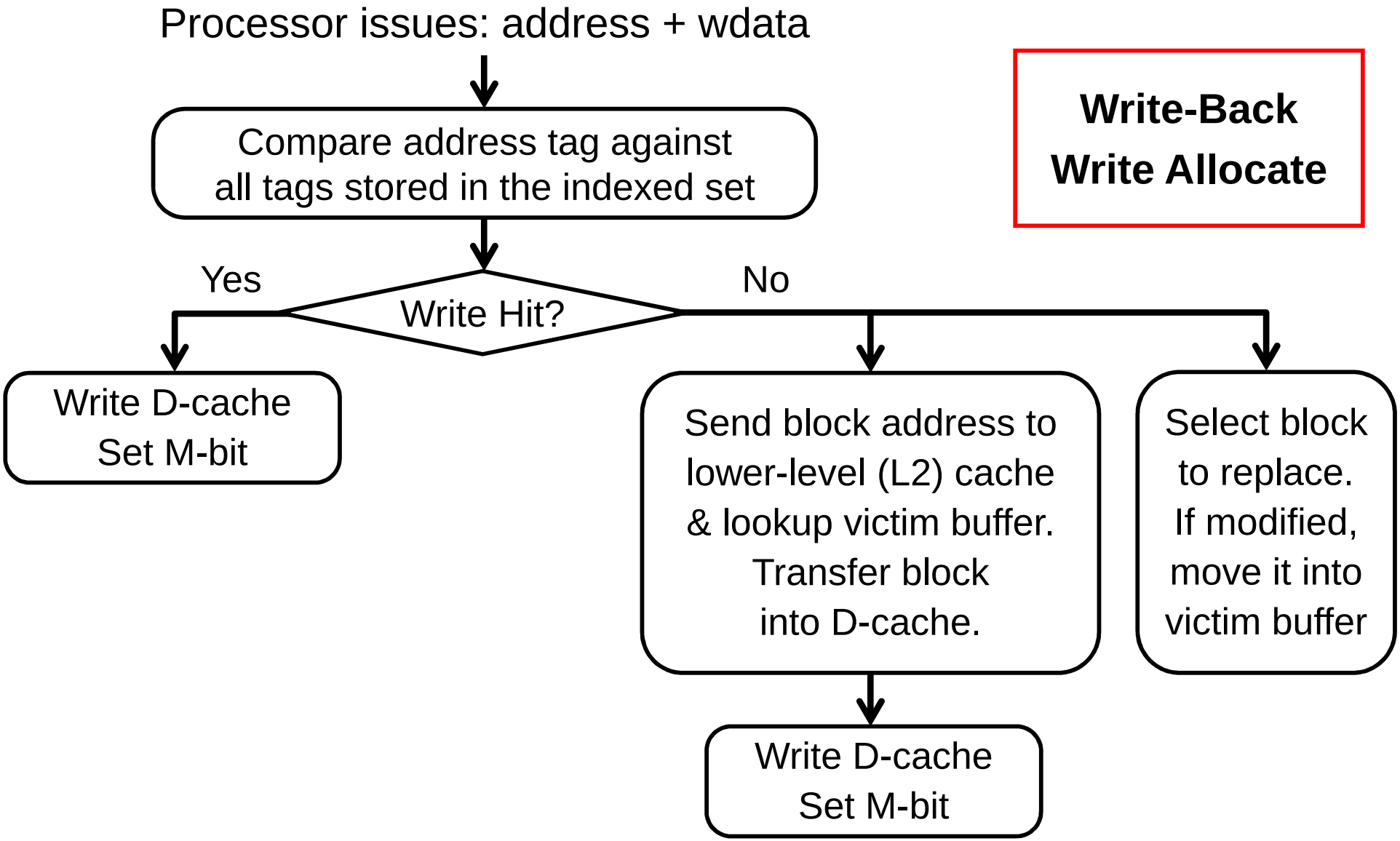


Write Through with Write No-Allocate

Write-Through
Write No-Allocate



Write Back with Write Allocate



Next . . .

- The Need for Cache Memory
- The Basics of Caches
- **Cache Performance**
- Multilevel Caches

Hit Rate and Miss Rate

- Hit Rate = Hits / (Hits + Misses)
- Miss Rate = Misses / (Hits + Misses)
- I-Cache Miss Rate = Miss rate in the Instruction Cache
- D-Cache Miss Rate = Miss rate in the Data Cache
- Example:
  - Out of 1000 instructions fetched, 60 missed in the I-Cache
  - 25% are load-store instructions, 50 missed in the D-Cache
  - What are the I-cache and D-cache miss rates?
- I-Cache Miss Rate = 60 / 1000 = 6%
- D-Cache Miss Rate = 50 / (25% × 1000) = 50 / 250 = 20%

Memory Stall Cycles

- The processor **stalls** on a Cache miss
  - When fetching instruction from I-Cache and Block is not present
  - When loading/storing data in a D-cache and Block is not present
  - When writing data in a write-though D-cache and write buffer is full
- Miss Penalty: clock cycles to process a cache miss

  Miss Penalty is assumed equal for I-cache & D-cache

  Miss Penalty is assumed equal for Load and Store
- Memory Stall Cycles =

  I-Cache Misses $\times$ Miss Penalty +

  D-Cache Read Misses $\times$ Miss Penalty +

  D-Cache Write Misses $\times$ Miss Penalty

Combined Misses

- Combined Misses =

  I-Cache Misses + D-Cache Read Misses + Write Misses

  I-Cache Misses = I-Count × I-Cache Miss Rate

  Read Misses = Load Count × D-Cache Read Miss Rate

  Write Misses = Store Count × D-Cache Write Miss Rate

- Combined misses are often reported per 1000 instructions

- Memory Stall Cycles = Combined Misses × Miss Penalty

Memory Stall Cycles Per Instruction

- Memory Stall Cycles Per Instruction =

  Combined Misses Per Instruction × Miss Penalty

  Miss Penalty is assumed equal for I-cache and D-cache

  Miss Penalty is assumed equal for Load and Store

- Combined Misses Per Instruction =

  I-Cache Miss Rate +

  Load Frequency × D-Cache Read Miss Rate +

  Store Frequency × D-Cache Write Miss Rate

Example on Memory Stall Cycles

- Consider a program with the given characteristics
  - 20% of instructions are load and 10% are store
  - I-cache miss rate is 2%
  - D-cache miss rate is 5% for load, and 1% for store
  - Miss penalty is 20 clock cycles for all cases (I-Cache & D-Cache)
  - Compute combined misses and stall cycles per instruction
- Combined misses per instruction in I-Cache and D-Cache
  - 2% + 20%×5% + 10%×1% = 0.031 combined misses per instruction
  - Equal to an average of 31 misses per 1000 instructions
- Memory stall cycles per instruction
  - 0.031×20 (miss penalty) = 0.62 memory stall cycles per instruction

CPU Time with Memory Stall Cycles

CPU Time =

(CPU execution cycles + Memory Stall Cycles) × Clock Cycle

$$\text{CPU Time} = \text{I-Count} \times \text{CPI}_{\text{overall}} \times \text{Clock Cycle}$$

$$\text{CPI}_{\text{overall}} = \text{CPI}_{\text{execution}} + \text{Memory Stall Cycles per Instruction}$$

- $\text{CPI}_{\text{overall}}$ = Overall CPI in the presence of cache misses
- $\text{CPI}_{\text{execution}}$ = Execution CPI (not counting cache misses)
- Memory stall cycles per instruction increases the overall CPI

Example on CPI with Memory Stalls

- A processor has $CPI_{execution}$ = 1.5 (not counting cache misses)
  - I-Cache miss rate is 2%, D-cache miss rate is 5% for load & store
  - 20% of instructions are loads and stores
  - Cache miss penalty is 100 clock cycles for I-cache and D-cache
- What is the impact of cache misses on the overall CPI?
- Answer: Memory Stall Cycles per Instruction =

  0.02×100 (I-Cache) + 0.2×0.05×100 (D-Cache) = 3

  $CPI_{overall}$ = 1.5 + 3 = 4.5 cycles per instruction

  $CPI_{overall}$ / $CPI_{execution}$ = 4.5 / 1.5 = 3

  Processor is 3 times slower due to memory stall cycles

Average Memory Access Time

- Average Memory Access Time (AMAT)

  AMAT = Hit time + Combined Miss rate × Miss penalty

  Hit Time = time to access the I-cache or D-cache (for a hit)

  Hit time is assumed to be the same for I-cache and D-cache

- Combined Miss Rate for Instruction Access and Data Access

$$\text{Combined Miss Rate} = \frac{\text{Combined Misses per Instruction}}{\text{Memory Accesses per Instruction}}$$

Memory Accesses per Instruction = 1 + %LS (%Load + %Store)

Miss penalty is assumed to be the same for I-cache and D-cache

AMAT Example

- Compute the overall average memory access time
  - Hit time = 1 clock cycle in both I-Cache and D-Cache
  - Miss penalty = 50 clock cycles for I-Cache and D-Cache
  - I-Cache misses = 3.8 misses per 1000 instructions
  - D-Cache misses = 41 misses per 1000 instructions
  - Load + Store frequency = 25%
- Solution:

Combined Misses per Instruction = (3.8 + 41) / 1000 = 0.0448

Combined Miss rate (per access) = 0.0448 / (1 + 0.25) = 0.03584

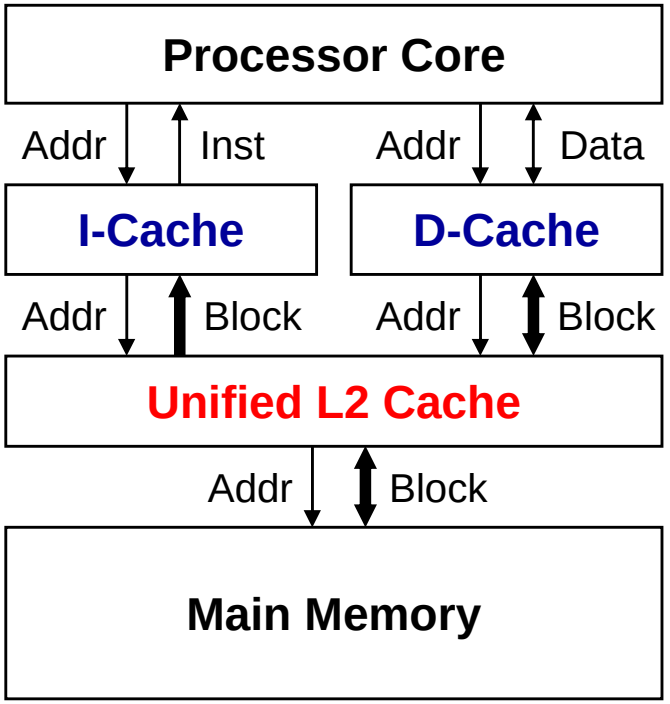
Overall AMAT = 1 + 0.03584 × 50 = 2.792 cycles

Next . . .

- The Need for Cache Memory
- The Basics of Caches
- Cache Performance
- **Multilevel Caches**

Multilevel Caches

- Top level cache is kept small to
  - Reduce hit time
  - Reduce energy per access
- Add another cache level to
  - Reduce the memory gap
  - Reduce memory bus loading
- Multilevel caches can help
  - Reduce miss penalty
  - Reduce average memory access time
- Large L2 cache can capture many misses in L1 caches
  - Reduce the global miss rate



For simplicity,
L3 cache is not included

Inclusion Policy

- Multi-Level Cache Inclusion
  - The content of the L1 cache is fully contained in the larger L2 cache
  - Wastes L2 cache space, but L2 has space for additional blocks
- Multi-Level Cache Exclusion
  - The L2 cache can only have blocks that are not in the L1 cache
  - Prevents wasting space
- Non-Inclusive Non-Exclusion (NINE)
  - Multi-level inclusion and exclusion must be enforced by a protocol
  - If neither is enforced then L2 cache is neither inclusive nor exclusive

Multilevel Inclusion

- L1 cache blocks are always present in L2 cache
  - Total number of unique cache blocks = L2 cache size
- A miss in L1, but a hit in L2 copies block from L2 to L1
- A miss in L1 and L2 brings a block into L1 and L2
- A write in L1 causes data to be written in L1 and L2
- Write-through policy is used from L1 to L2
- Write-back policy is used from L2 to lower-level memory
  - To reduce traffic on the memory bus
- A replacement (or invalidation) in L2 must be seen in L1
  - A block which is evicted from L2 must also be evicted from L1

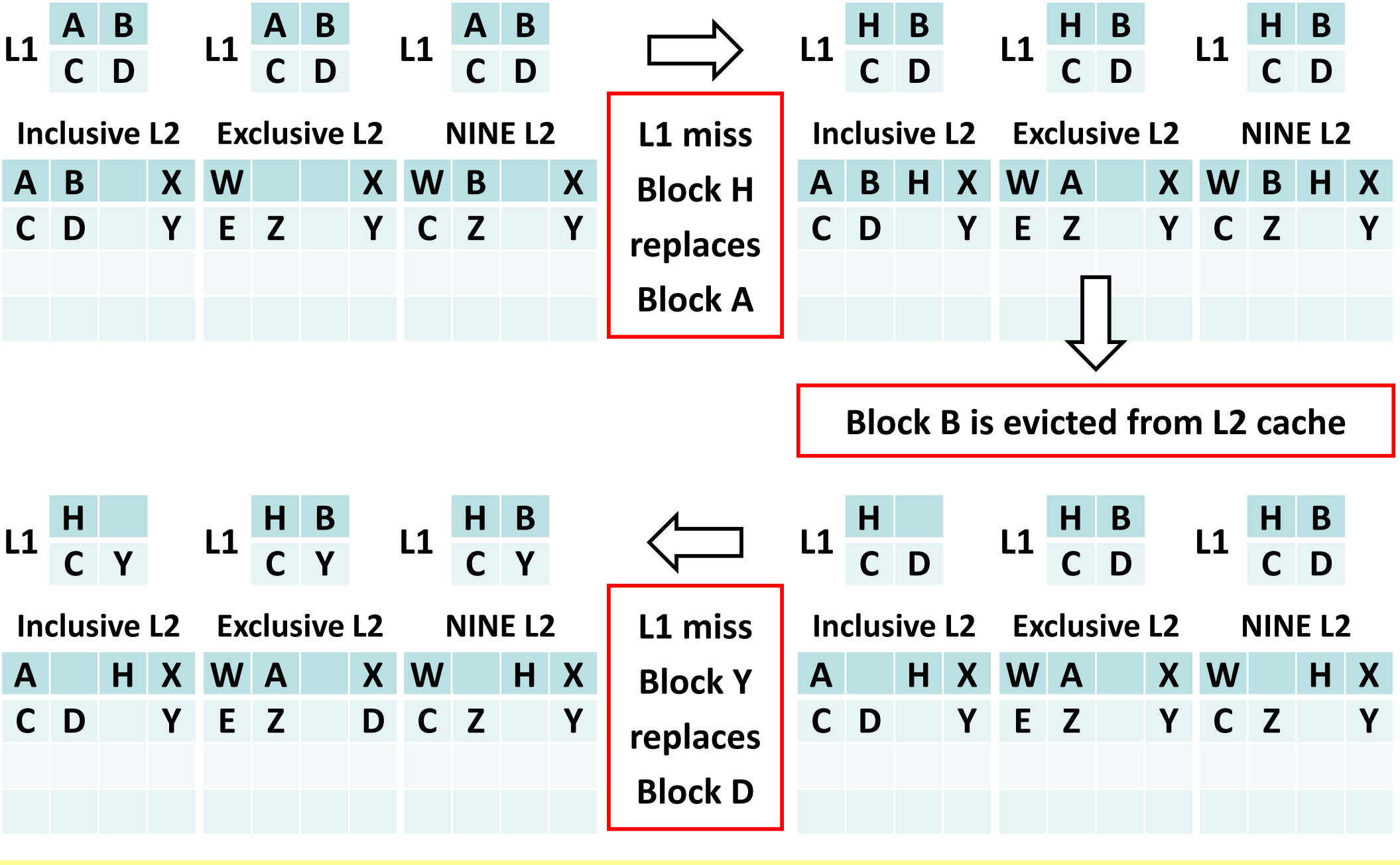
Multilevel Exclusion

- L1 cache blocks are never found in L2 cache
  - Prevents wasting space
  - Total number of unique cache blocks = L1 + L2 cache size
- Cache miss in L1 and L2 brings the block into L1 only
- Cache miss in L1, but hit in L2 results in a swap of blocks
  - More complex to implement
- Block replaced in L1 is moved into L2
  - L2 cache stores L1 evicted blocks, in case needed later in L1
  - L2 cache acts as a **victim cache**
- Write-Back policy from L1 to L2
- Write-Back policy from L2 to lower-level memory

Non-Inclusive, Non Exclusive (NINE)

- L1 cache blocks may or may not be present in L2 cache
  - Total number of unique cache blocks > L2 cache size, but < (L1 + L2) size
- Cache miss in L1 and L2 brings a block into L1 and L2
- A miss in L1, but a hit in L2 copies block from L2 to L1
  - A block replaced in L1 can be discarded, if not modified
  - If a block is replaced and modified in L1, then it must be written back to L2
- A block can be replaced in L2 without removing it from L1
  - If L2 is inclusive, then block evicted from L2 must also be evicted from L1
- Write-back policy between L1 and L2
  - Write-through is not possible if a block exists in L1, but replaced in L2

Inclusive, Exclusive, and NINE L2 Cache



Local and Global Miss Rates

❖ Local Miss Rate

Number of cache misses / Memory accesses to this cache

Miss Rate$_{L1}$ for L1 cache

Miss Rate$_{L2}$ for L2 cache

❖ Global Miss Rate

Number of cache misses / Memory accesses generated by processor

MissRate$_{L1}$ for L1 cache (same as local)

MissRate$_{L1}$ $\times$ MissRate$_{L2}$ for L2 cache (different from local)

Global miss rate is a better measure for L2 cache

Fraction of the total memory accesses that miss in L1 and L2

Example of Local & Global Miss Rates

- **Problem:** suppose that out of 1000 instructions
  - 5 instructions missed in the I-Cache and 39 missed in the D-Cache
  - 15 instructions missed in the L2 cache
  - Load + Store instruction frequency = 30%
- Compute: L1 miss rate, L2 local and global miss rates
- **Solution:**

L1 misses per instruction = (5 + 39) / 1000 = 0.044

Memory accesses per instruction = 1 + 0.3 = 1.3

L1 Miss Rate = 0.044 / 1.3 = 0.0338 misses per access

L2 Miss Rate = 15 / 44 = 0.341 misses per L2 access

L2 Global Miss Rate = 0.0338 × 0.341 = 0.0115 (equal to 15/1000/1.3)

Average Memory Access Time

$$AMAT = Hit\ time_{L1} + Miss\ Rate_{L1} \times Miss\ Penalty_{L1}$$

$$Hit\ time_{L1} = Hit\ time_{I\text{-}Cache} = Hit\ time_{D\text{-}Cache}$$

$$Miss\ Penalty_{L1} = Miss\ Penalty_{I\text{-}Cache} = Miss\ Penalty_{D\text{-}Cache}$$

$$Miss\ Rate_{L1} = Combined\ Misses\ per\ Instruction_{L1} / (1 + \%LS)$$

Miss Penalty in the presence of L2 Cache

$$Miss\ Penalty_{L1} = Hit\ time_{L2} + Miss\ Rate_{L2} \times Miss\ Penalty_{L2}$$

AMAT in the presence of L2 Cache

$$AMAT = Hit\ time_{L1} +$$

$$Miss\ Rate_{L1} \times (Hit\ time_{L2} + Miss\ Rate_{L2} \times Miss\ Penalty_{L2})$$

AMAT Example with L2 Cache

- **Problem:** Compute Average Memory Access Time
  - I-Cache Miss Rate = 1%, D-Cache Miss Rate = 10%
  - L2 Cache Miss Rate = 40%
  - L1 Hit time = 1 clock cycle (identical for I-Cache and D-Cache)
  - L2 Hit time = 8 cycles, L2 Miss Penalty = 100 cycles
  - Load + Store instruction frequency = 25%
- **Solution:**
  - Memory Access per Instruction = 1 + 25% = 1.25
  - $\text{Misses per Instruction}_{L1}$ = 1% + 25% × 10% = 0.035
  - Miss Rate_{L1} = 0.035 / 1.25 = 0.028 (2.8%)
  - Miss Penalty_{L1} = 8 + 0.4 × 100 = 48 cycles
  - AMAT = 1 + 0.028 × 48 = 2.344 cycles

Memory Stall Cycles Per Instruction

Memory Stall Cycles per Instruction

$= \text{Misses per Instruction}_{L1} \times \text{Miss Penalty}_{L1}$

$= \text{Memory Access per Instruction} \times \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$

$= (1 + \%LS) \times \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})$

Memory Stall Cycles per Instruction

$= \text{Misses per Instruction}_{L1} \times \text{Hit Time}_{L2} +$

$\text{Misses per Instruction}_{L2} \times \text{Miss Penalty}_{L2}$

$\text{Misses per Instruction}_{L1} = (1 + \%LS) \times \text{Miss Rate}_{L1}$

$\text{Misses per Instruction}_{L2} = (1 + \%LS) \times \text{Miss Rate}_{L1} \times \text{Miss Rate}_{L2}$

Two-Level Cache Performance

- **Problem:** out of 1000 addresses generated by program
  - I-Cache misses = 5, D-Cache misses = 35, L2 Cache misses = 8
  - L1 Hit = 1 cycle, L2 Hit = 8 cycles, L2 Miss penalty = 80 cycles
  - Load + Store frequency = 25%, $CPI_{execution}$ = 1.1
  - Compute memory stall cycles per instruction and effective CPI
  - If the L2 Cache is removed, what will be the effective CPI?
- **Solution:**
  - L1 Miss Rate = (5 + 35) / 1000 = 0.04 (or 4% per access)
  - L1 misses per Instruction = 0.04 × (1 + 0.25) = 0.05
  - L2 misses per Instruction = (8 / 1000) × 1.25 = 0.01
  - Memory stall cycles per Instruction = 0.05 × 8 + 0.01 × 80 = 1.2
  - CPI_{L1+L2} = 1.1 + 1.2 = 2.3, CPI/$CPI_{execution}$ = 2.3/1.1 = 2.1x slower
  - CPI_{L1only} = 1.1 + 0.05 × 80 = 5.1 (worse)