Computer Architecture

Memory Hierarchy: Set-Associative Cache

Nouman M Durrani

Courtesy of Prof. Hong Jiang, Yifeng Zhu (U. Maine)

Spring 2019

Cache performance

- **Miss-oriented Approach to Memory Access:**

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty \right) \times CycleTime$$

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemMisses}{Inst} \times MissPenalty \right) \times CycleTime$$

  - $CPI_{Execution}$ includes ALU and Memory instructions

- **Separating out Memory component entirely**
  - AMAT = Average Memory Access Time
  - CPI_{ALUOps} does not include memory instructions

$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

$$\begin{aligned} AMAT &= HitTime + MissRate \times MissPenalty \\ &= \left(HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst} \right) + \\ &\quad \left(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data} \right) \end{aligned}$$

Cache Performance Example

- **Assume we have a computer where the clock per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2% (Unified instruction cache and data cache), how much faster would the computer be if all instructions and data were cache hit?**

$$CPUtime = \left(CPUClockCycles + MemeoryStalls\right) \times ClockCycleTime$$
$$= (IC \times CPI + MemoryStalls) \times ClockCycleTime$$

When all instructions are hit

$$CPUtime\_Ideal = (IC \times CPI + MemoryStalls) \times ClockCycleTime$$
$$= (IC \times 1.0 + 0) \times ClockCycleTime$$
$$= IC \times ClockCycleTime$$

In reality:

$$MemoryStallCycles = IC \times \frac{MemAccess}{Inst} \times MissRate \times MissPenalty$$
$$= IC \times (1 + 0.5) \times 0.02 \times 25 = IC \times 0.75$$

$$CPUtime\_Cache = (IC \times CPI + MemoryStalls) \times ClockCycleTime$$
$$= (IC \times 1.0 + IC \times 0.75) \times ClockCycleTime$$
$$= 1.75 \times IC \times ClockCycleTime$$

Performance Example Problem

Assume:

- **For gcc, the frequency for all loads and stores is 36%.**
- **instruction cache miss rate for gcc = 2%**
- **data cache miss rate for gcc = 4%.**
- **If a machine has a CPI of 2 without memory stalls**
- **and the miss penalty is 40 cycles for all misses,**

how much faster is a machine with a perfect cache?

Instruction miss cycles =IC x 2% x 40 = 0.80 x IC

Data miss cycles = IC x 36% x 4% x 40 = 0.576 x IC

CPIstall = 2 + (0.80 + 0.567) = 2 + 1.376 = 3.376

$$\frac{\text{IC x CPIstall x Clock period}}{\text{IC x CPIperfect x Clock period}} = \frac{3.376}{2} = 1.69$$

Performance Example Problem

Assume: we increase the performance of the previous machine by doubling its clock rate. Since the main memory speed is unlikely to change, assume that the absolute time to handle a cache miss does not change. How much faster will the machine be with the faster clock?

For gcc, the frequency for all loads and stores is 36%

Instruction miss cycles	**= IC x 2% x 80**	**= 1.600 x IC**
Data miss cycles	**= IC x 36% x 4% x 80**	**= <u>1.152 x IC</u>**
		2.752 x IC

$$\frac{I \times CPI_{slowClk} \times \text{Clock period}}{I \times CPI_{fastClk} \times \text{Clock period}} = \frac{3.376}{4.752 \times 0.5} = 1.42 \text{ (not 2)}$$

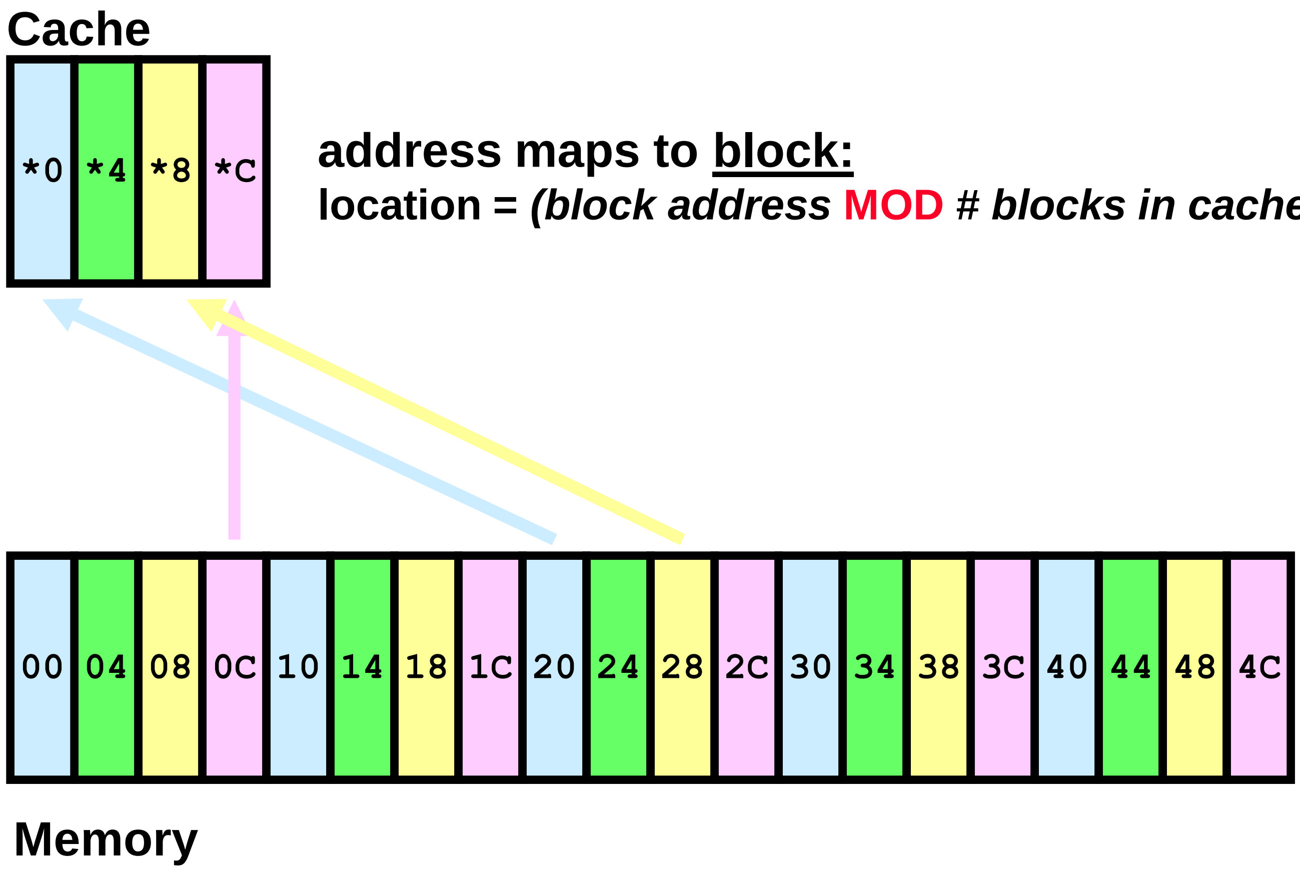
Fundamental Questions

- **Q1: Where can a block be placed in the upper level?** ***(Block placement)***
- **Q2: How is a block found if it is in the upper level?** ***(Block identification)***
- **Q3: Which block should be replaced on a miss?** ***(Block replacement)***
- **Q4: What happens on a write?** ***(Write strategy)***

Q1: Block Placement

- Where can block be placed in cache?
  - In one predetermined place - direct-mapped
    - Use part of address to calculate block location in cache
    - Compare cache block with tag to check if block present
  - Anywhere in cache - fully associative
    - Compare tag to every block in cache
  - In a limited set of places - set-associative
    - Use portion of address to calculate set (like direct-mapped)
    - Place in any block in the set
    - Compare tag to every block in set
    - Hybrid of direct mapped and fully associative

Direct Mapped Block Placement



address maps to <u>block</u>:
location = *(block address* MOD *# blocks in cache)*

Example: Accessing A Direct-Mapped Cache

- **DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

DM Memory Access 1: Mapping: 0 modulo 4 = 0

Mem Block	DM Hit/Miss
0	

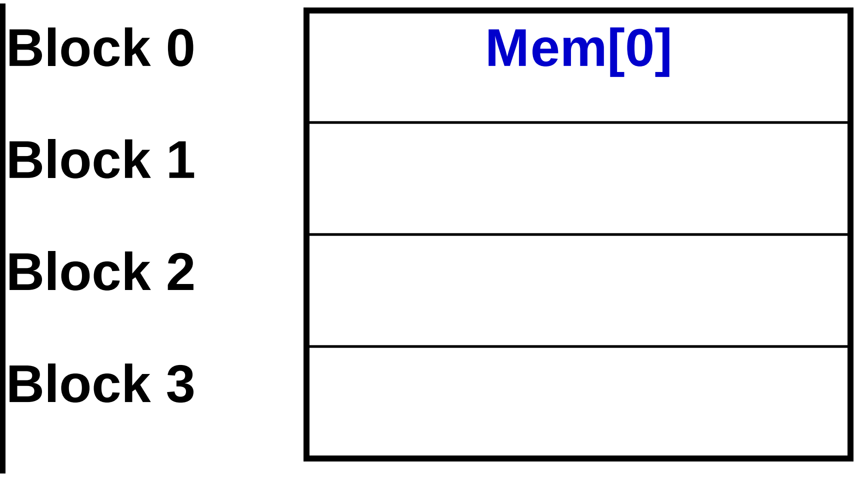
Block 0	
Block 1	
Block 2	
Block 3	

Example: Accessing A Direct-Mapped Cache

DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

DM Memory Access 1: Mapping: 0 mod 4 = 0

Mem Block	DM Hit/Miss
0	miss



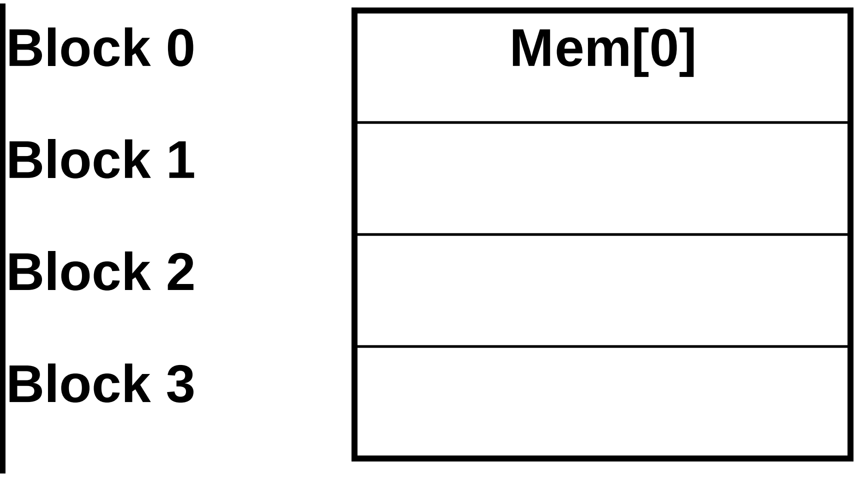
Set 0 is empty: write Mem[0]

Example: Accessing A Direct-Mapped Cache

- DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

DM Memory Access 2: Mapping: 8 mod 4 = 0

Mem Block	DM Hit/Miss
0	miss
8	



Example: Accessing A Direct-Mapped Cache

- **DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

DM Memory Access 2: Mapping: 8 mod 4 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss

Block 0	Mem[8]
Block 1	
Block 2	
Block 3	

Set 0 contains Mem[0]. Overwrite with Mem[8]

Example: Accessing A Direct-Mapped Cache

- **DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

DM Memory Access 3: Mapping: 0 mod 4 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss
0	

Block 0	Mem[8]
Block 1	
Block 2	
Block 3	

Example: Accessing A Direct-Mapped Cache

- **DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

DM Memory Access 3: Mapping: 0 mod 4 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss
0	miss

Block 0	Mem[0]
Block 1	
Block 2	
Block 3	

Set 0 contains Mem[8]. Overwrite with Mem[0]

Example: Accessing A Direct-Mapped Cache

- DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

DM Memory Access 4: Mapping: 6 mod 4 = 2

Mem Block	DM Hit/Miss
0	miss
8	miss
0	miss
6	

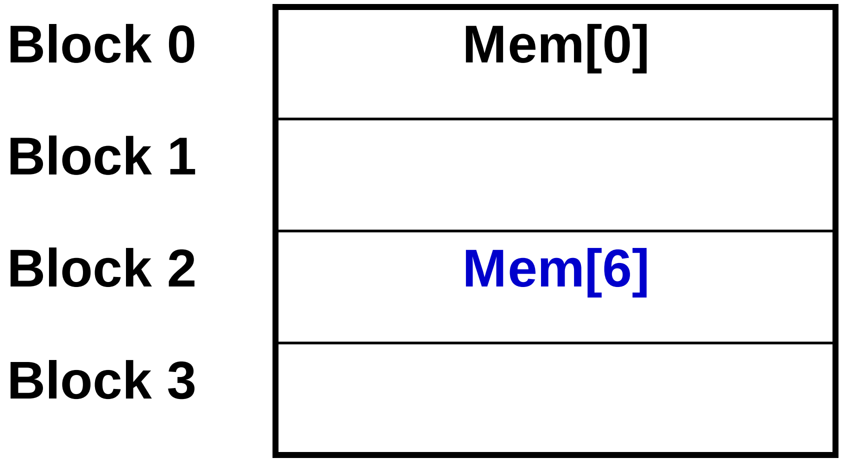
Block 0	Mem[0]
Block 1	
Block 2	
Block 3	

Example: Accessing A Direct-Mapped Cache

- DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

DM Memory Access 4: Mapping: 6 mod 4 = 2

Mem Block	DM Hit/Miss
0	miss
8	miss
0	miss
6	miss



Set 2 empty. Write Mem[6]

Example: Accessing A Direct-Mapped Cache

- DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

DM Memory Access 5: Mapping: 8 mod 4 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss
0	miss
6	miss
8	

Block 0	Mem[0]
Block 1	
Block 2	Mem[6]
Block 3	

Example: Accessing A Direct-Mapped Cache

- DM cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

DM Memory Access 5: Mapping: 8 mod 4 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss
0	miss
6	miss
8	miss

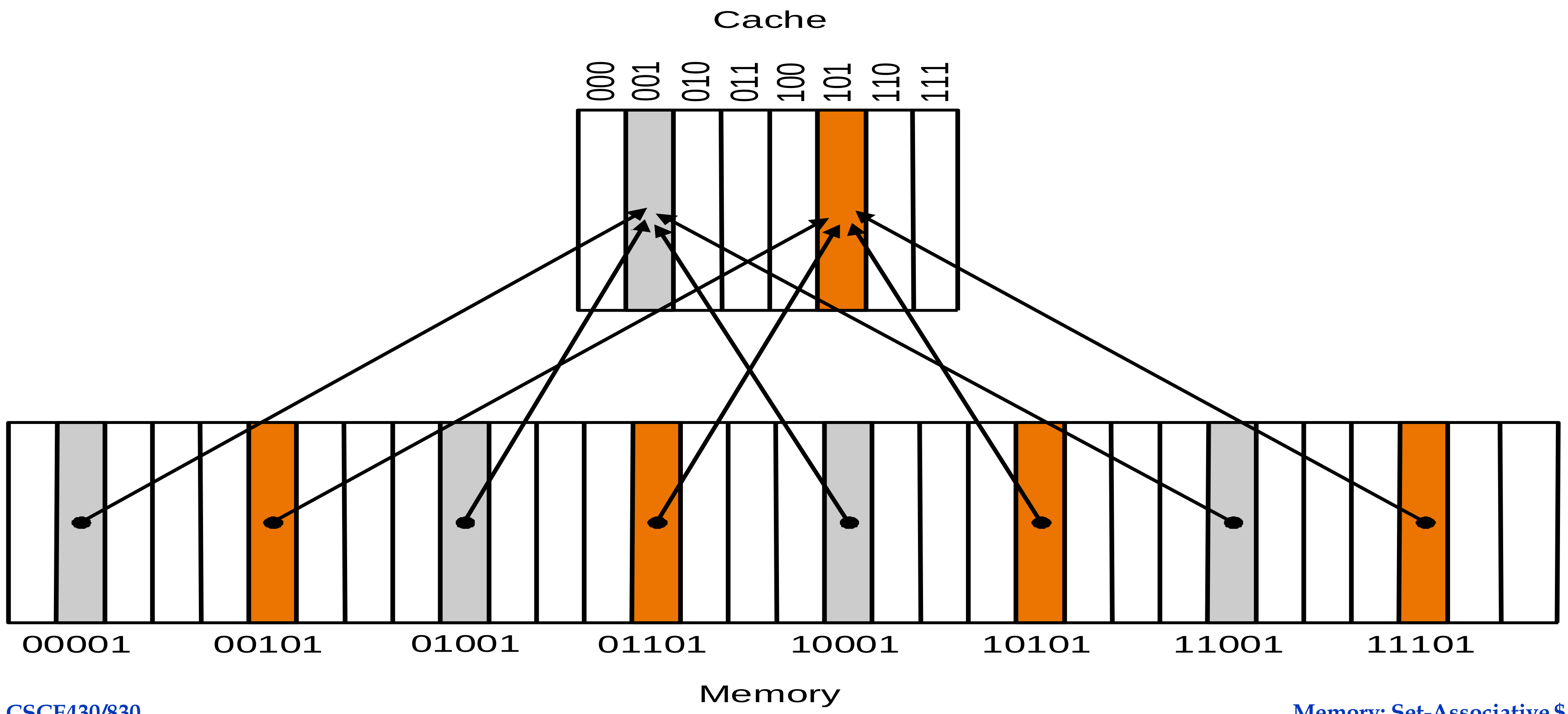
Block 0	Mem[8]
Block 1	
Block 2	Mem[6]
Block 3	

Set 0 contains Mem[0]. Overwrite with Mem[8]

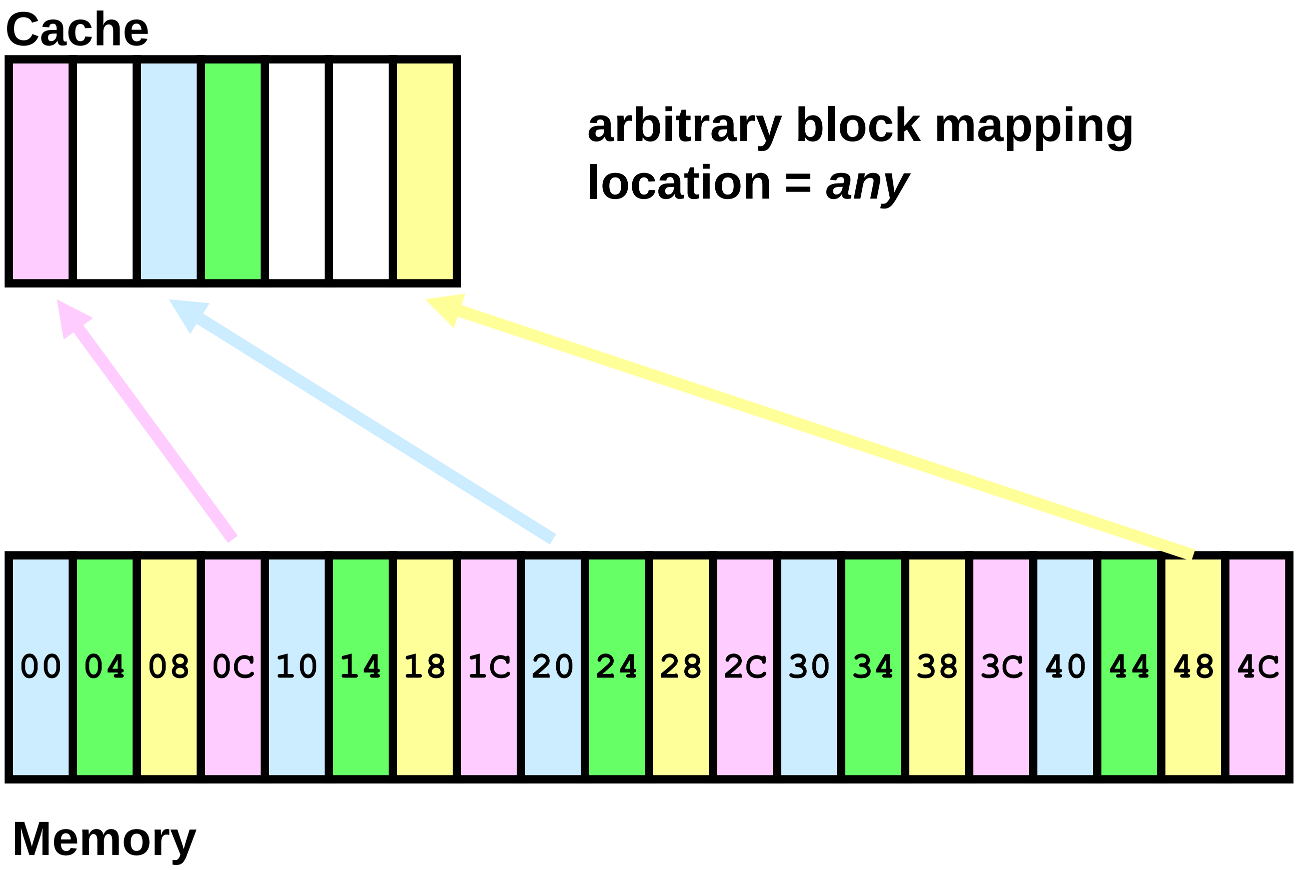
Direct-Mapped Cache with n one-word blocks

- Pros: find data fast
- Con: What if access 00001 and 10001 repeatedly?

→ We always miss…



Fully Associative Block Placement



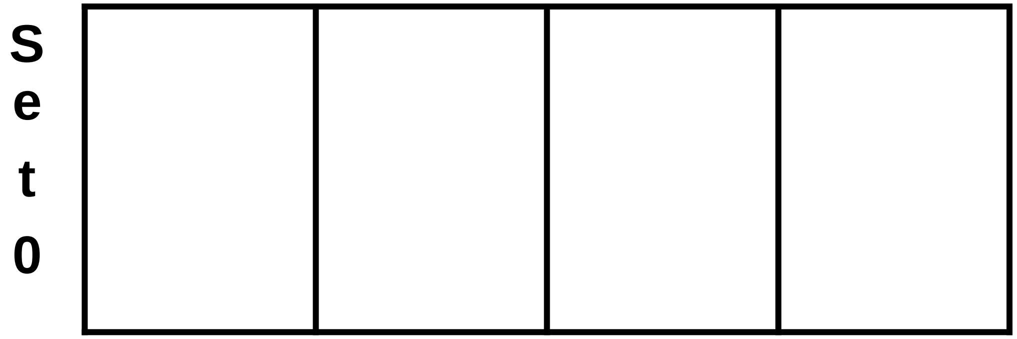
arbitrary block mapping
location = *any*

Example: Accessing A Fully-Associative Cache

- **Fully-Associative cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

FA Memory Access 1:

Mem Block	DM Hit/Miss
0	



FA Block Replacement Rule: replace least recently used block in set

Example: Accessing A Fully-Associative Cache

- **Fully-Associative cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

FA Memory Access 1:

Mem Block	DM Hit/Miss
0	miss

Set 0	Mem [0]			

Set 0 is empty: write Mem[0] to Block 0

Example: Accessing A Fully-Associative Cache

- **Fully-Associative cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

FA Memory Access 2:

Mem Block	DM Hit/Miss
0	miss
8	

Set 0	Mem [0]			

Example: Accessing A Fully-Associative Cache

- Fully-Associative cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

FA Memory Access 2:

Mem Block	DM Hit/Miss
0	miss
8	miss

Set 0				
	Mem [0]	Mem [8]		

Blocks 1-3 are LRU: write Mem[8] to Block 1

Example: Accessing A Fully-Associative Cache

- **Fully-Associative cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

FA Memory Access 3:

Mem Block	DM Hit/Miss
0	miss
8	miss
0	

Set 0	Mem [0]	Mem [8]		

Example: Accessing A Fully-Associative Cache

- **Fully-Associative cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

FA Memory Access 3:

Mem Block	DM Hit/Miss
0	miss
8	miss
0	hit

Set 0	Mem [0]	Mem [8]		

Block 0 contains Mem[0]

Example: Accessing A Fully-Associative Cache

- Fully-Associative cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

FA Memory Access 4:

Mem Block	DM Hit/Miss
0	miss
8	miss
0	hit
6	

Set 0	Mem [0]	Mem [8]		

Example: Accessing A Fully-Associative Cache

- **Fully-Associative cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

FA Memory Access 4:

Mem Block	DM Hit/Miss
0	miss
8	miss
0	hit
6	miss

Set 0	Mem [0]	Mem [8]	Mem [6]	

Blocks 2-3 are LRU : write Mem[6] to Block 2

Example: Accessing A Fully-Associative Cache

- **Fully-Associative cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

FA Memory Access 5:

Mem Block	DM Hit/Miss
0	miss
8	miss
0	hit
6	miss
8	

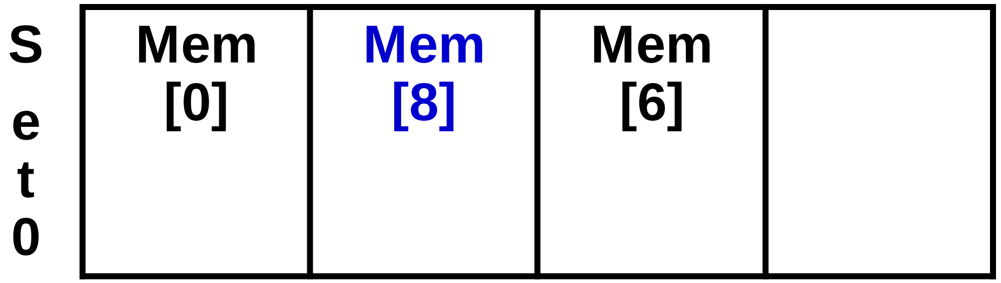
Set 0	Mem [0]	Mem [8]	Mem [6]	

Example: Accessing A Fully-Associative Cache

- **Fully-Associative cache contains 4 1-word blocks. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

FA Memory Access 5:

Mem Block	DM Hit/Miss
0	miss
8	miss
0	hit
6	miss
8	hit

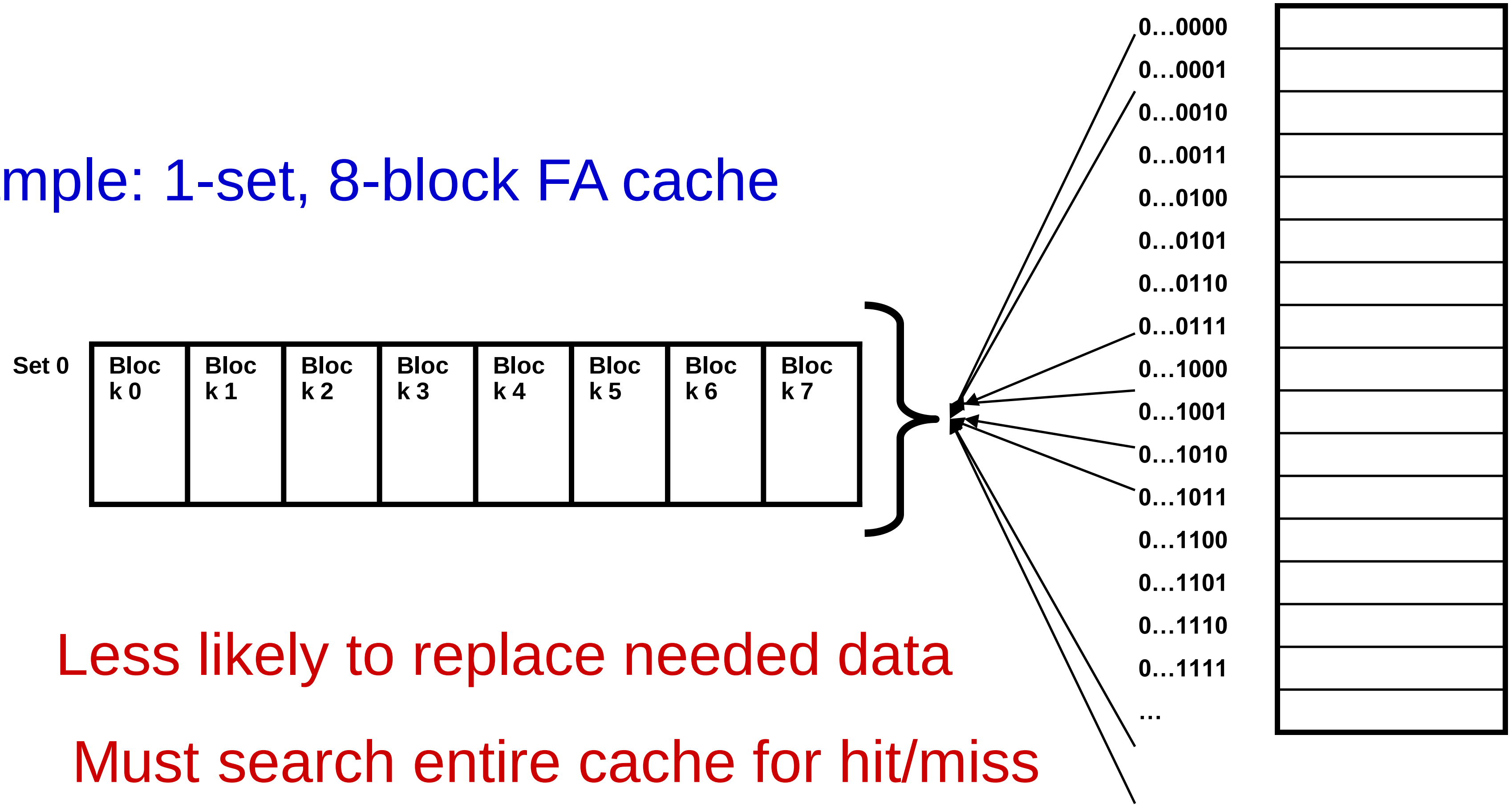


Block 1 contains Mem[8]

Fully-Associative Cache Basics

1 set, n blocks: no mapping restrictions on how blocks are stored in cache: many ways, e.g. least recently used is replaced (LRU)

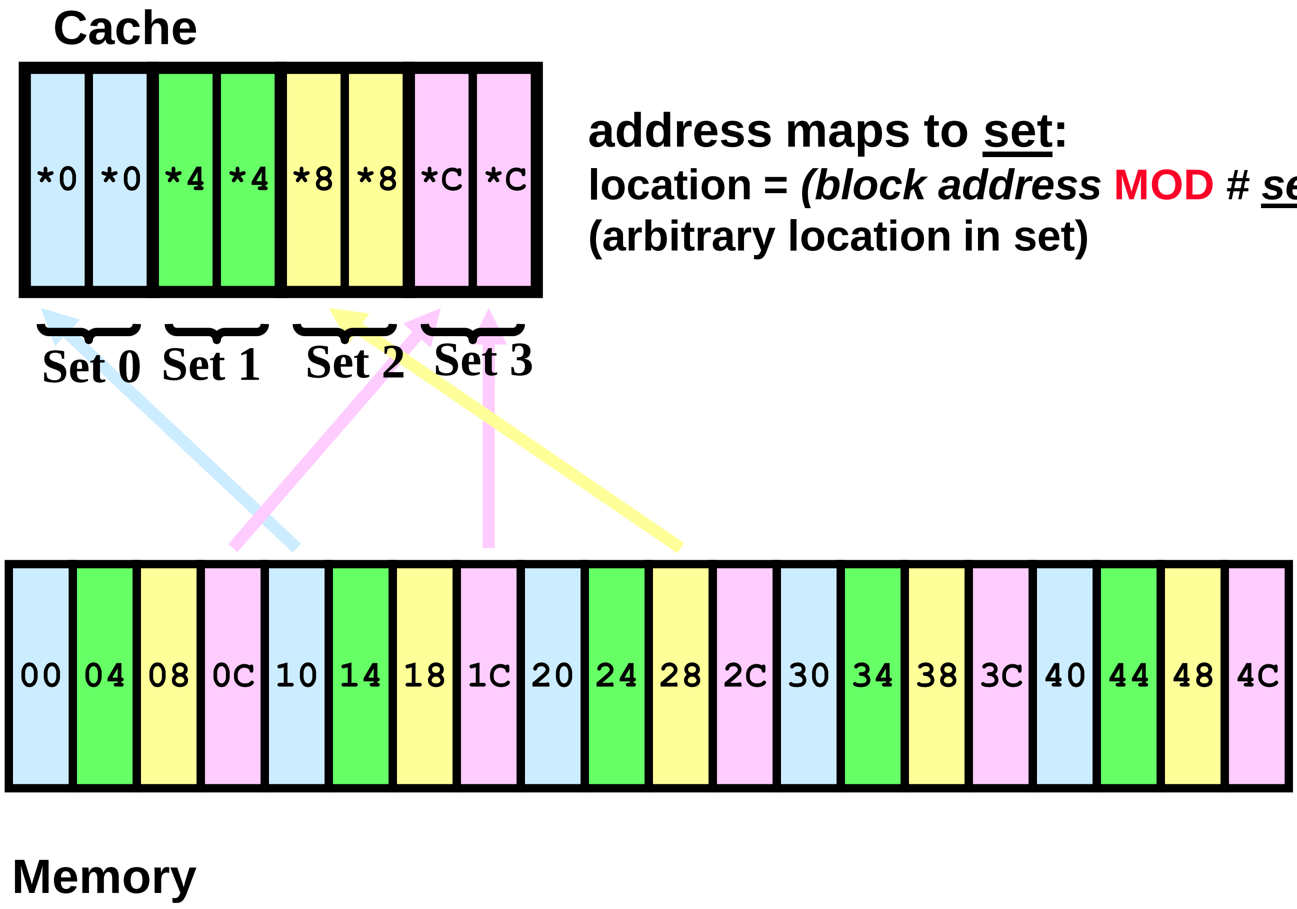
Example: 1-set, 8-block FA cache



PRO: Less likely to replace needed data

CON: Must search entire cache for hit/miss

Set-Associative Block Placement



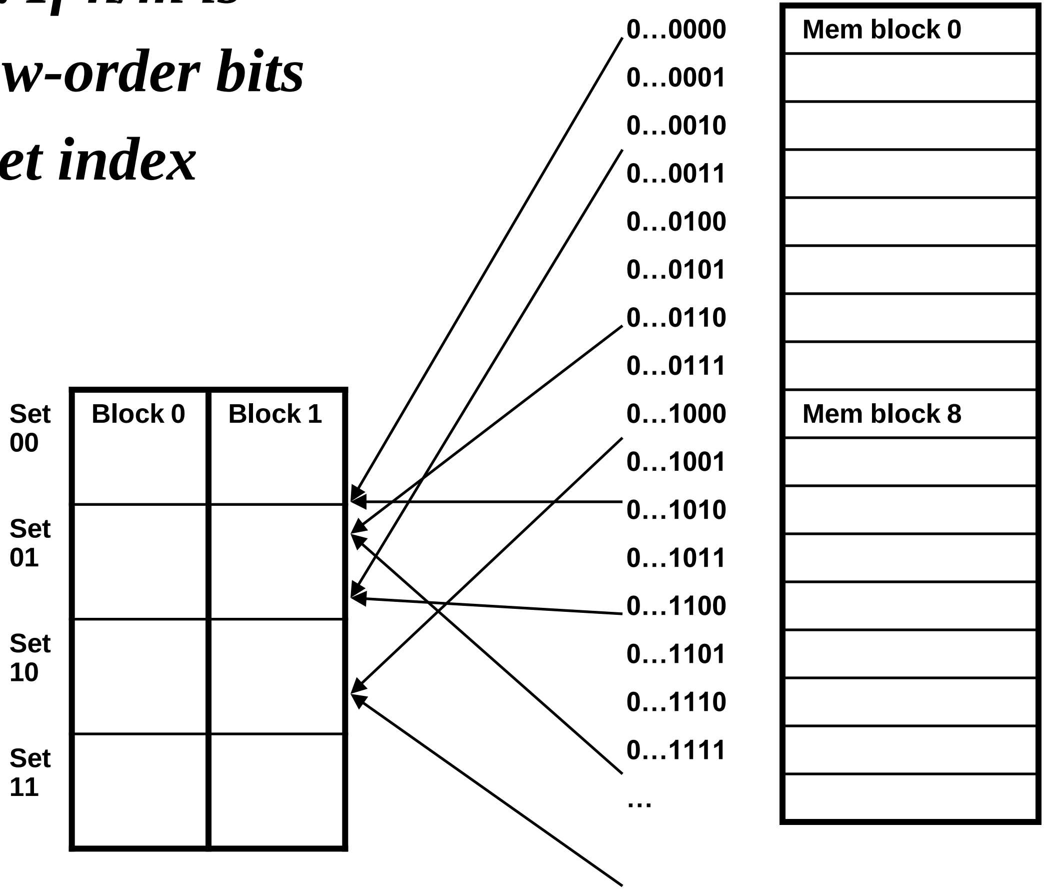
address maps to set:
location = *(block address* **MOD** # *sets in cache*)
(arbitrary location in set)

Set-Associative Cache Basics

n/m sets, m blocks (m-way): blocks are mapped from memory location to a specific set in cache

Mapping: Mem Address % n/m. If n/m is a power of 2, log2(n/m) = #low-order bits of memory address = cache set index

Example: 4 set, 2-way SA cache (ADD mod 4)



Example: Accessing A Set-Associative Cache

- **2-way Set-Associative cache contains 2 sets, 2 one-word blocks each. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

SA Memory Access 1: Mapping: 0 mod 2 = 0

Mem Block	DM Hit/Miss
0	

Set 0		
Set 1		

SA Block Replacement Rule: replace least recently used block in set

Example: Accessing A Set-Associative Cache

- 2-way Set-Associative cache contains 2 sets, 2 one-word blocks each. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

SA Memory Access 1: Mapping: 0 mod 2 = 0

Mem Block	DM Hit/Miss
0	miss

Set 0	Mem[0]	
Set 1		

Set 0 is empty: write Mem[0] to Block 0

Example: Accessing A Set-Associative Cache

- 2-way Set-Associative cache contains 2 sets, 2 one-word blocks each. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

SA Memory Access 2: Mapping: 8 mod 2 = 0

Mem Block	DM Hit/Miss
0	miss
8	

Set 0	Mem[0]	
Set 1		

Example: Accessing A Set-Associative Cache

- 2-way Set-Associative cache contains 2 sets, 2 one-word blocks each. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

SA Memory Access 2: Mapping: 8 mod 2 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss

Set 0	Mem[0]	Mem[8]
Set 1		

Set 0, Block 1 is LRU: write Mem[8]

Example: Accessing A Set-Associative Cache

- 2-way Set-Associative cache contains 2 sets, 2 one-word blocks each. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

SA Memory Access 3: Mapping: 0 mod 2 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss
0	

Set 0	Mem[0]	Mem[8]
Set 1		

Example: Accessing A Set-Associative Cache

- **2-way Set-Associative cache contains 2 sets, 2 one-word blocks each. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

SA Memory Access 3: Mapping: 0 mod 2 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss
0	hit

Set 0	Mem[0]	Mem[8]
Set 1		

Set 0, Block 0 contains Mem[0]

Example: Accessing A Set-Associative Cache

- 2-way Set-Associative cache contains 2 sets, 2 one-word blocks each. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

SA Memory Access 4: Mapping: 6 mod 2 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss
0	hit
6	

Set 0	Mem[0]	Mem[8]
Set 1		

Example: Accessing A Set-Associative Cache

- **2-way Set-Associative cache contains 2 sets, 2 one-word blocks each. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

SA Memory Access 4: Mapping: 6 mod 2 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss
0	hit
6	miss

Set 0	Mem[0]	Mem[6]
Set 1		

Set 0, Block 1 is LRU: overwrite with Mem[6]

Example: Accessing A Set-Associative Cache

- 2-way Set-Associative cache contains 2 sets, 2 one-word blocks each. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8

SA Memory Access 5: Mapping: 8 mod 2 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss
0	hit
6	miss
8	

Set 0	Mem[0]	Mem[6]
Set 1		

Example: Accessing A Set-Associative Cache

- **2-way Set-Associative cache contains 2 sets, 2 one-word blocks each. Find the # Misses for each cache given this sequence of memory block accesses: 0, 8, 0, 6, 8**

SA Memory Access 5: Mapping: 8 mod 2 = 0

Mem Block	DM Hit/Miss
0	miss
8	miss
0	hit
6	miss
8	miss

Set 0	Mem[8]	Mem[6]
Set 1		

Set 0, Block 0 is LRU: overwrite with Mem[8]

Set-Associative Cache Basics

n/m sets, m blocks (m-way): blocks are mapped from memory location to a specific set in cache

Mapping:* *Mem Address % n/m. If n/m is a power of 2, log2(n/m) = #low-order bits of memory address = cache set index

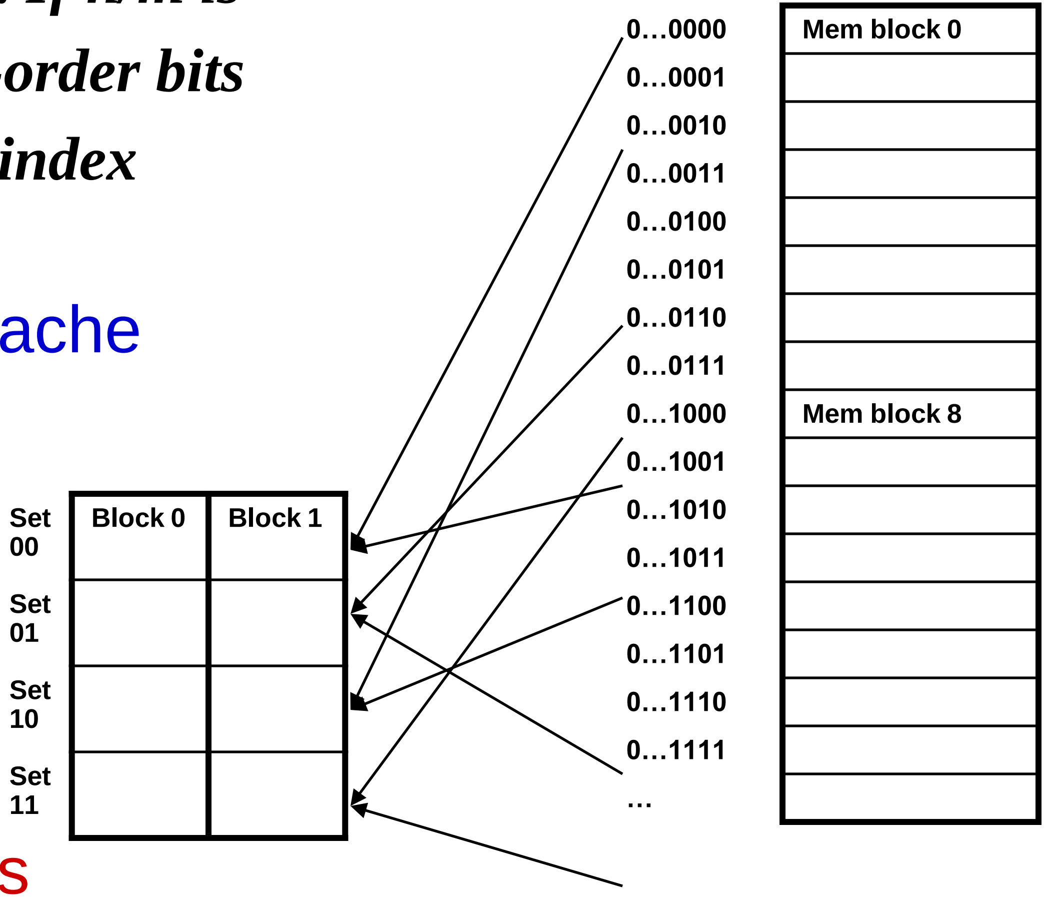
Example: 4 set, 2-way SA cache (X mod 4)

PRO:

Easier to find but won't always overwrite

CON:

Must search set for hit/miss

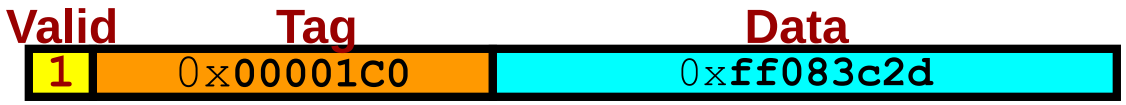


Associativity Considerations

- DM and FA are special cases of SA cache
  - Set-Associative: n/m sets; m blocks/set (associativity=m)
  - Direct-Mapped: m=1 (1-way set-associative, associativity=1)
  - Fully-Associative: m=n (n-way set-associative, associativity=n)
- Advantage of Associativity: as associativity increases, miss rate decreases (because more blocks per set that we’re less likely to overwrite)
- Disadvantage of Associativity: as associativity increases, hit time increases (because we have to search more blocks – more HW required)
- Block Replacement: LRU or random. Random is easier to implement and often not much worse

Q2: Block Identification

- **Every cache block has an address tag that identifies its location in memory**
- **Hit when tag and address of desired word match (comparison by hardware)**
- **Q: What happens when a cache block is empty?**
  A: Mark this condition with a valid bit (0 if empty- 0 if the data is not in the cache otherwise 1)



Q2: Block Identification?

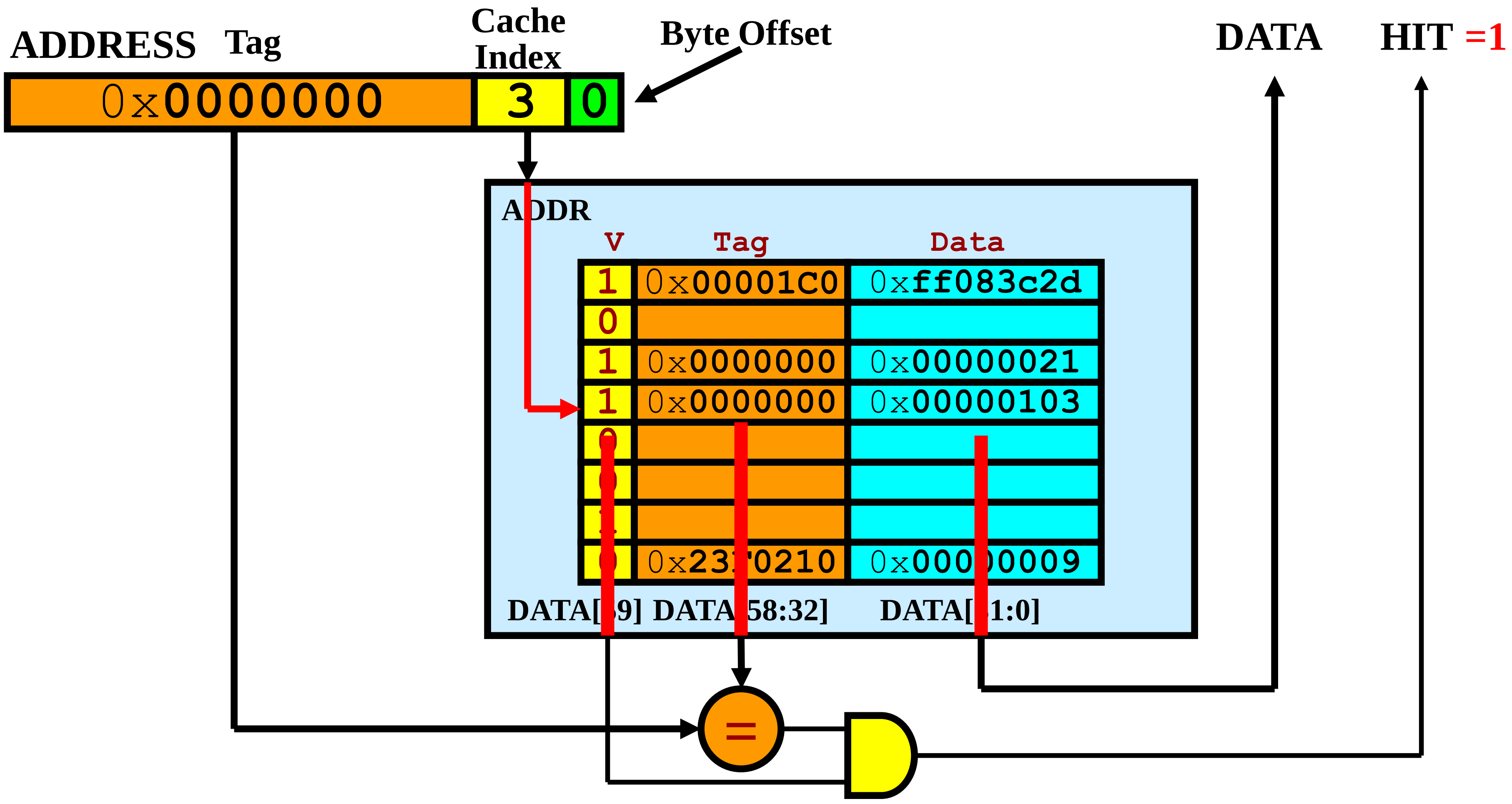
- **Tag on each block**
  - **No need to check index or block offset**
- **Increasing associativity shrinks index, expands tag**

Block Address		Block Offset
Tag	Index	

Fully Associative: No index

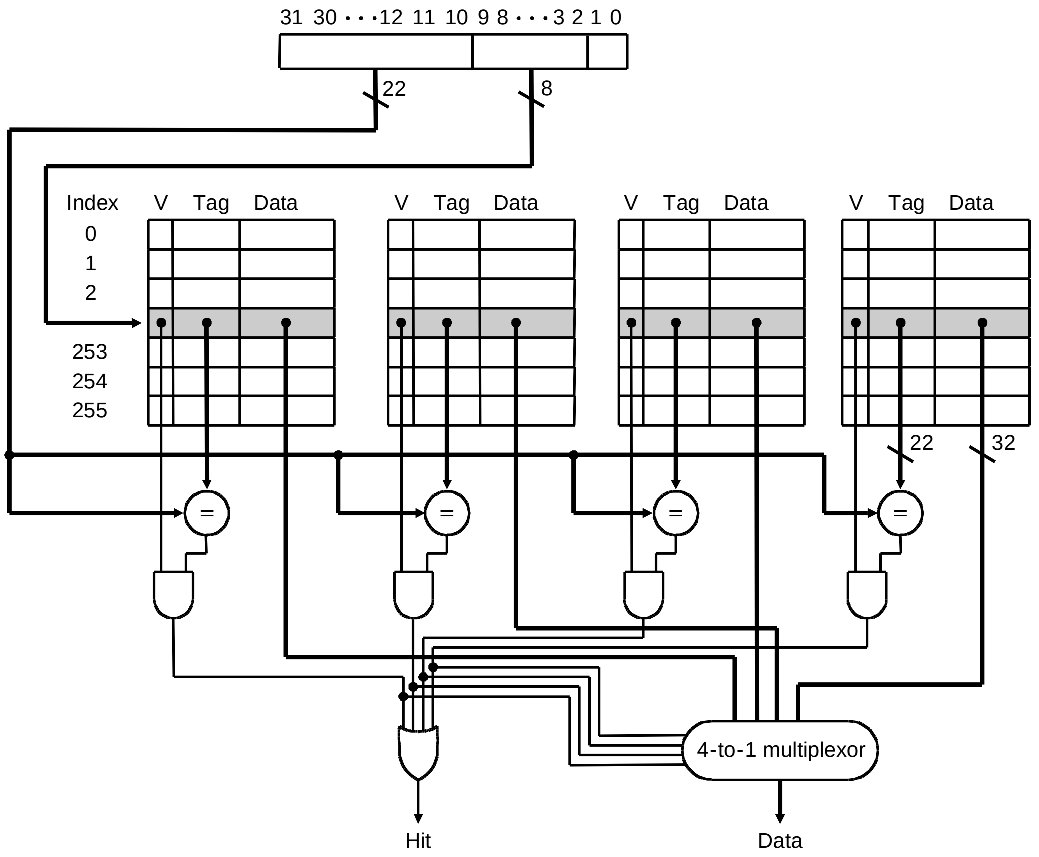
Direct Mapped: Large index

Direct-Mapped Cache Design



Set Associative Cache Design

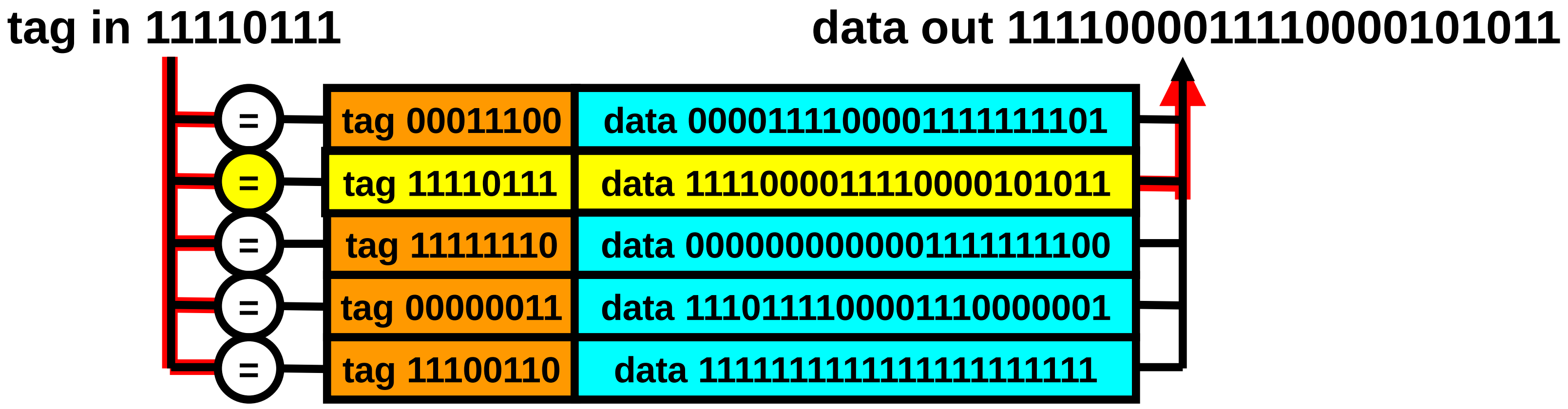
- **Key idea:**
  - **Divide cache into <u>sets</u>**
  - **Allow block <u>anywhere</u> in a set**
- **Advantages:**
  - **Better hit rate**
- **Disadvantage:**
  - **More tag bits**
  - **More hardware**
  - **Higher access time**



A Four-Way Set-Associative Cache

Fully Associative Cache Design

- Key idea: set size of one block
  - 1 comparator required for each block
  - No address decoding
  - Practical only for small caches due to hardware demands



tag in 11110111
data out 1111000011110000101011
=
tag 00011100
data 0000111100001111111101
=
tag 11110111
data 1111000011110000101011
=
tag 11111110
data 0000000000001111111100
=
tag 00000011
data 1110111100001110000001
=
tag 11100110
data 1111111111111111111111

Calculating Bits in Cache

- How many total bits are needed for a direct- mapped cache with 64 KBytes of data and one word blocks, assuming a 32-bit address?

- How many total bits would be needed for a 4-way set associative cache to store the same amount of data

- How many total bits are needed for a direct- mapped cache with 64 KBytes of data and 8 word blocks, assuming a 32-bit address?

Calculating Bits in Cache

- How many total bits are needed for a direct- mapped cache with 64 KBytes of data. Consider a 4 byte block size and a 32-bit addressing.
  - 64 Kbytes = 16 K words = 2^14 words = 2^14 blocks
  - block size = 4 bytes => offset size = 2 bits,
  - #blocks = 2^14 => index size = 14 bits
  - tag size = address size - index size - offset size = 32 - 14 - 2 = 16 bits
  - bits/block = data bits + tag bits + valid bit = 32 + 16 + 1 = 49
  - bits in cache = #blocks x bits/block = 2^14 x 49 = 98 Kbytes
- How many total bits would be needed for a 4-way set associative cache to store the same amount of data
  - block size and #blocks does not change
  - #sets = #blocks/4 = (2^14)/4 = 2^12 => index size = 12 bits
  - tag size = address size - index size - offset = 32 - 12 - 2 = 18 bits
  - bits/block = data bits + tag bits + valid bit = 32 + 18 + 1 = 51
  - bits in cache = #blocks x bits/block = 2^14 x 51 = 102 Kbytes
- Increase associativity => increase bits in cache

Calculating Bits in Cache

- How many total bits are needed for a direct- mapped cache with 64 KBytes of data and 8 word blocks, assuming a 32-bit address?
  - 64 Kbytes = 2^14 words = (2^14)/8 = 2^11 blocks
  - block size = 32 bytes => offset size = 5 bits,
  - #sets = #blocks = 2^11 => index size = 11 bits
  - tag size = address size - index size - offset size = 32 - 11 - 5 = 16 bits
  - bits/block = data bits + tag bits + valid bit = 8x32 + 16 + 1 = 273 bits
  - bits in cache = #blocks x bits/block = 2^11 x 273 = 68.25 Kbytes
- Increase block size => decrease bits in cache

Q3: Block Replacement

- On a miss, data must be read from memory.
- So, where do we put the new data?
  - Direct-mapped cache: must place in fixed location
  - Set-associative, fully-associative - can pick within set

Replacement Algorithms

- ***When a block is fetched, which block in the target set should be replaced?***
- **Optimal algorithm:**
  - » **replace the block that will not be used for the longest time (must know the future)**
- **Usage based algorithms:**
  - – **Least recently used (LRU)**
    - » **replace the block that has been referenced least recently**
    - » **hard to implement**
- **Non-usage based algorithms:**
  - – **First-in First-out (FIFO)**
    - » **treat the set as a circular queue, replace head of queue.**
    - » **easy to implement**
  - – **Random (RAND)**
    - » **replace a random block in the set**
    - » **even easier to implement**

Q4: Write Strategy

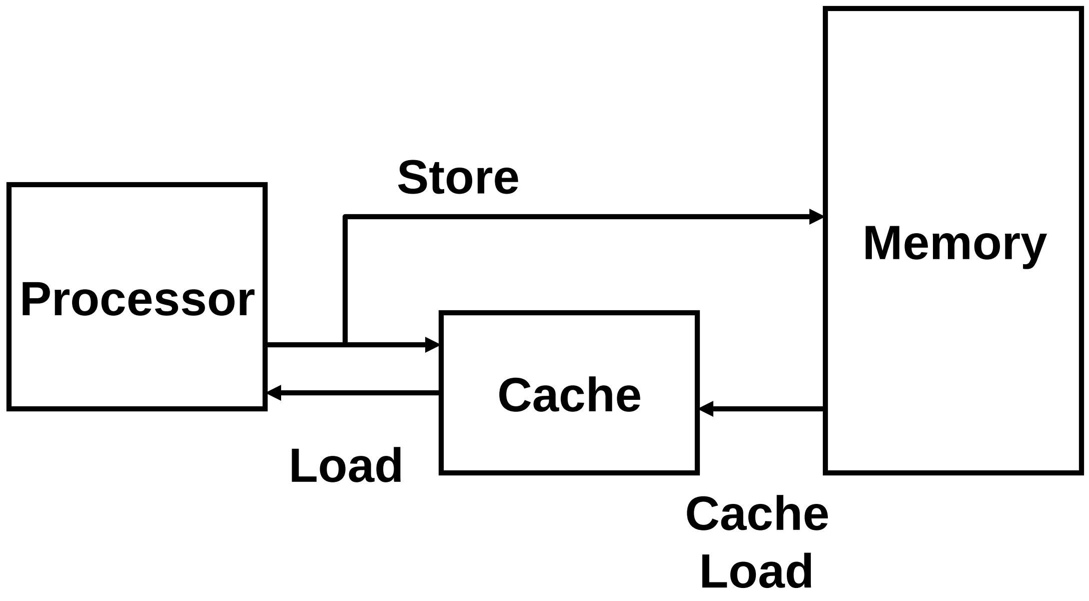
- What happens on a write?
  - Write through - write to memory, stall processor until done
  - Write buffer - place in buffer (allows pipeline to continue*)
  - Write back - delay write to memory until block is replaced in cache
- Special considerations when using DMA, multiprocessors (coherence between caches)

Q4: Write Strategy

- **What happens on a write?**
  - **Write through - write to memory, stall processor until done**
  - **Write buffer - place in buffer (allows pipeline to continue*)**
  - **Write back - delay write to memory until block is replaced in cache**

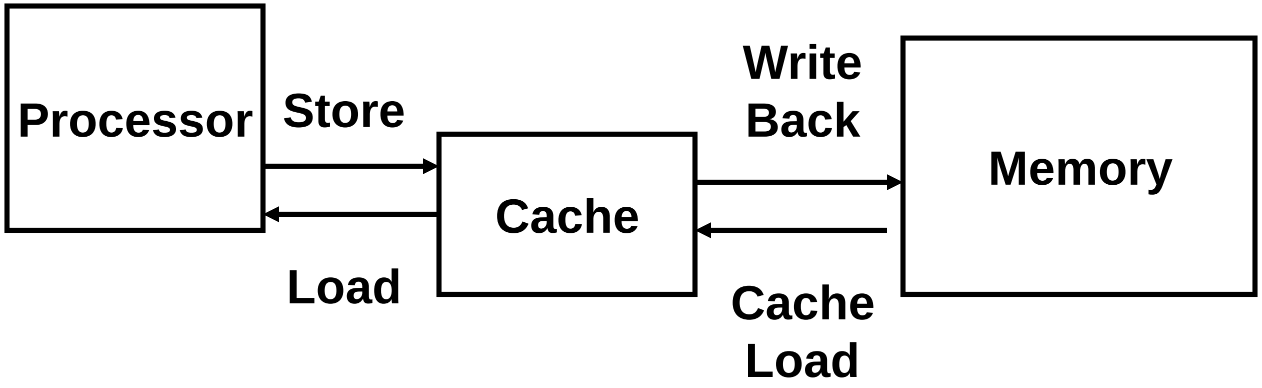
Write Through

- Store by processor updates cache *and* memory
- Memory always consistent with cache
- ~2X more loads than stores
- WT always combined with write buffers so that don’t wait for lower level memory



Write Back

- Store by processor only updates cache line
- Modified line written to memory only when it is evicted
  - Requires "dirty bit" for each line
    - » Set when line in cache is modified
    - » Indicates that line in memory is stale
- Memory not always consistent with cache
- No writes of repeated writes



Store Miss?

- Write-Allocate
  - Bring written block into cache
  - Update word in block
  - Anticipate further use of block
- No-write Allocate
  - Main memory is updated
  - Cache contents unmodified

Cache Basics

- **Cache:** level of temporary memory storage between CPU and main memory. Improves overall memory speed by taking advantage of the principle of locality
- Cache is divided into **sets**; each set holds from a particular group of main memory locations
- Cache parameters
  - Cache size, block size, associativity
- 3 types of Cache (w/ n total blocks):
  - Direct-mapped: n sets, each holds 1 block
  - Fully-associative: 1 set, holds n blocks
  - Set-associative: n/m sets, each holds m blocks

Classifying Misses: 3C

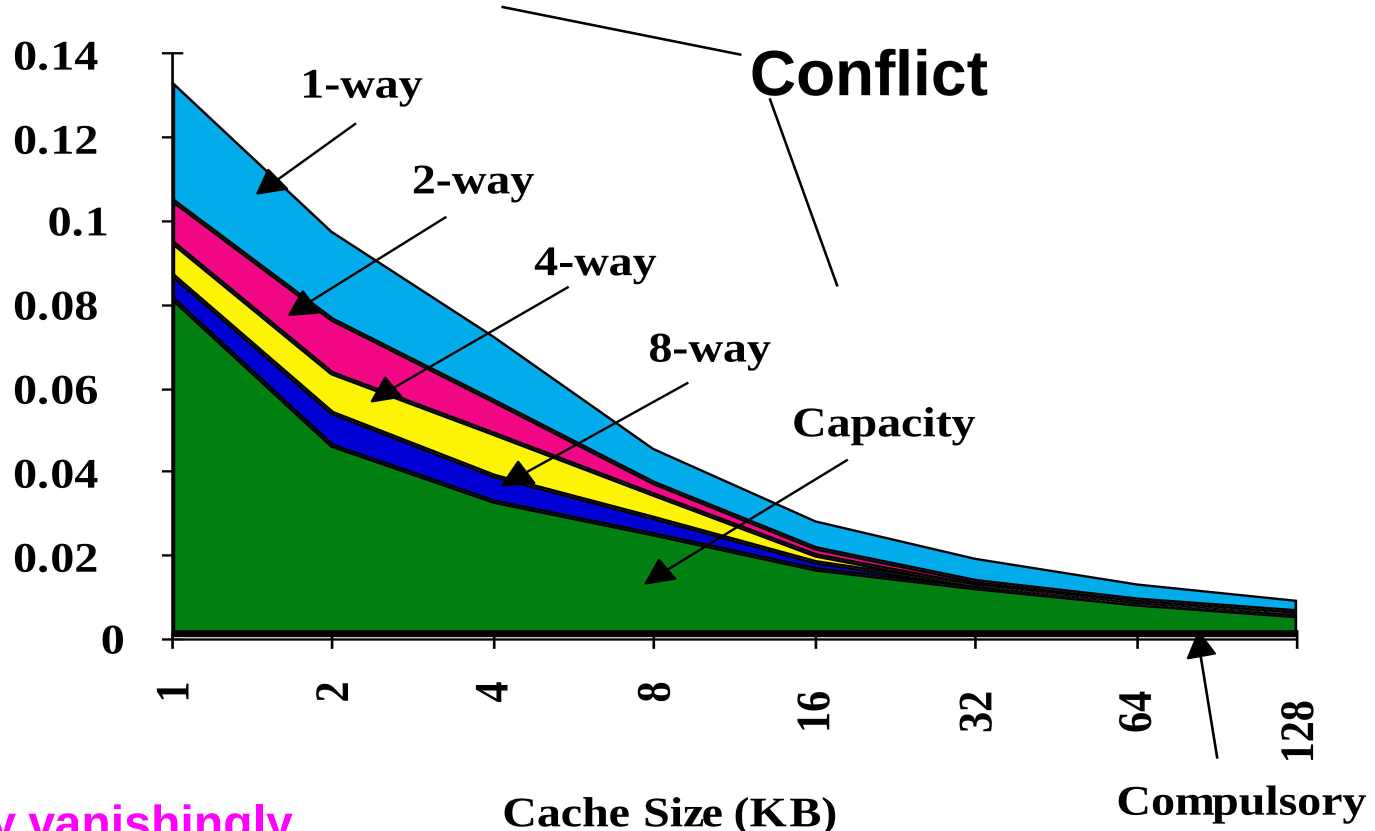
– *Compulsory*—The first access to a block is not in the cache, so the block must be brought into the cache. Also called *cold start misses* or *first reference misses*.
(Misses in even an Infinite Cache)

– *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.
(Misses in Fully Associative Size X Cache)

– *Conflict*—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called *collision misses* or *interference misses*.
(Misses in N-way Associative, Size X Cache)

Classifying Misses: 3C

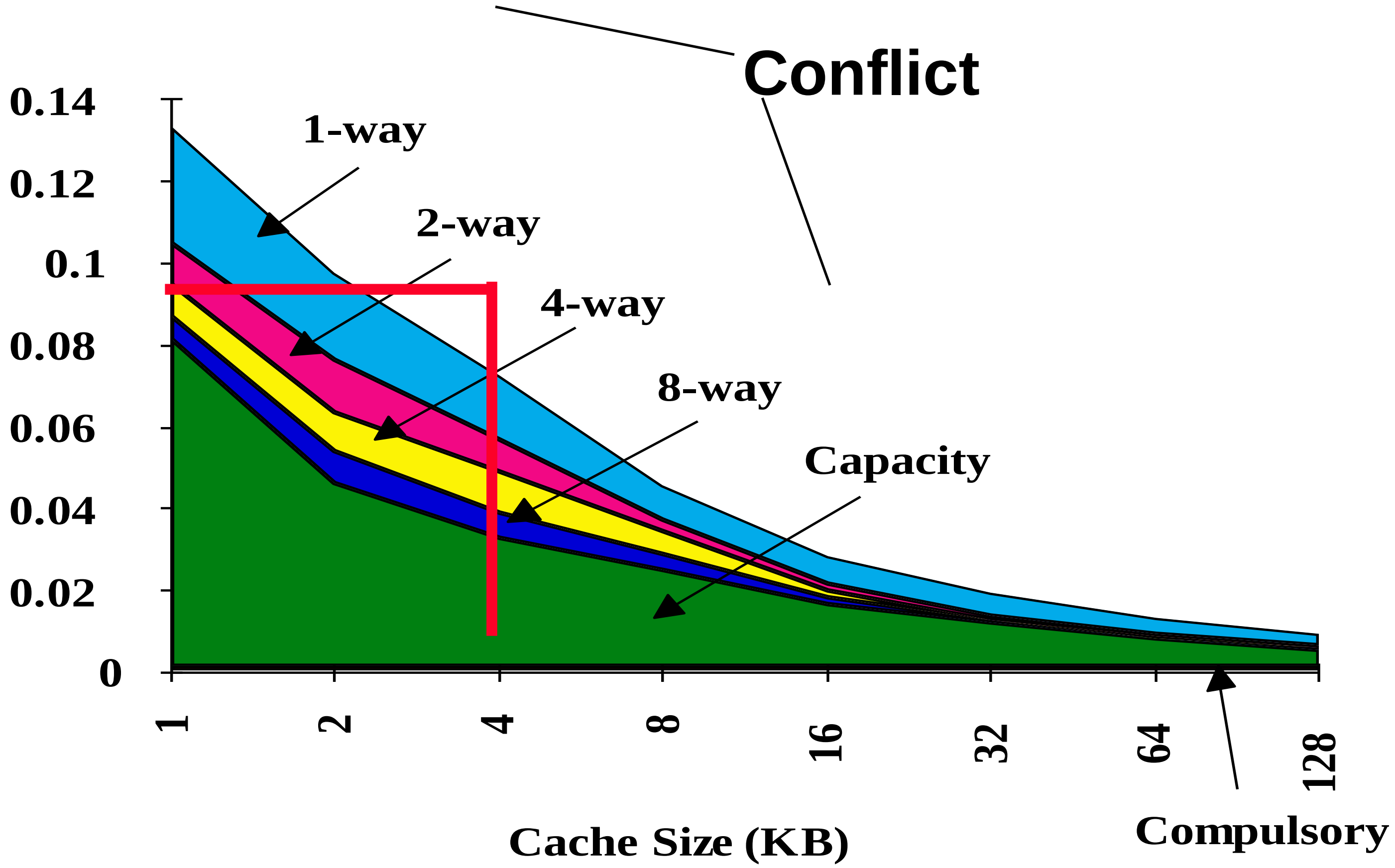
3Cs Absolute Miss Rate (SPEC92)



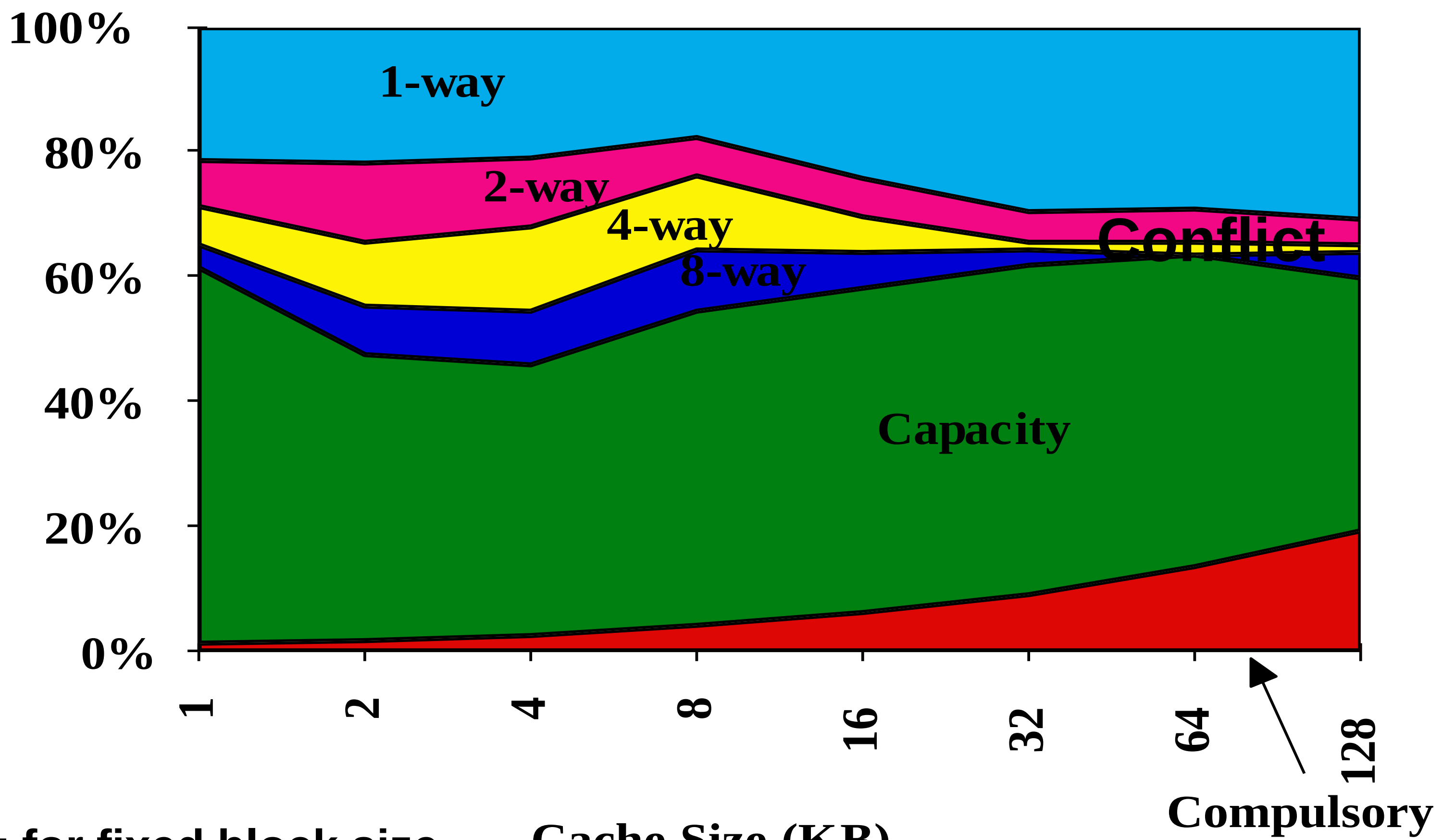
Compulsory vanishingly small

2:1 Cache Rule

miss rate 1-way associative cache size X
= miss rate 2-way associative cache size X/2



3C Relative Miss Rate



Flaws: for fixed block size
Good: insight => invention

Improve Cache Performance

improve cache and memory access times:

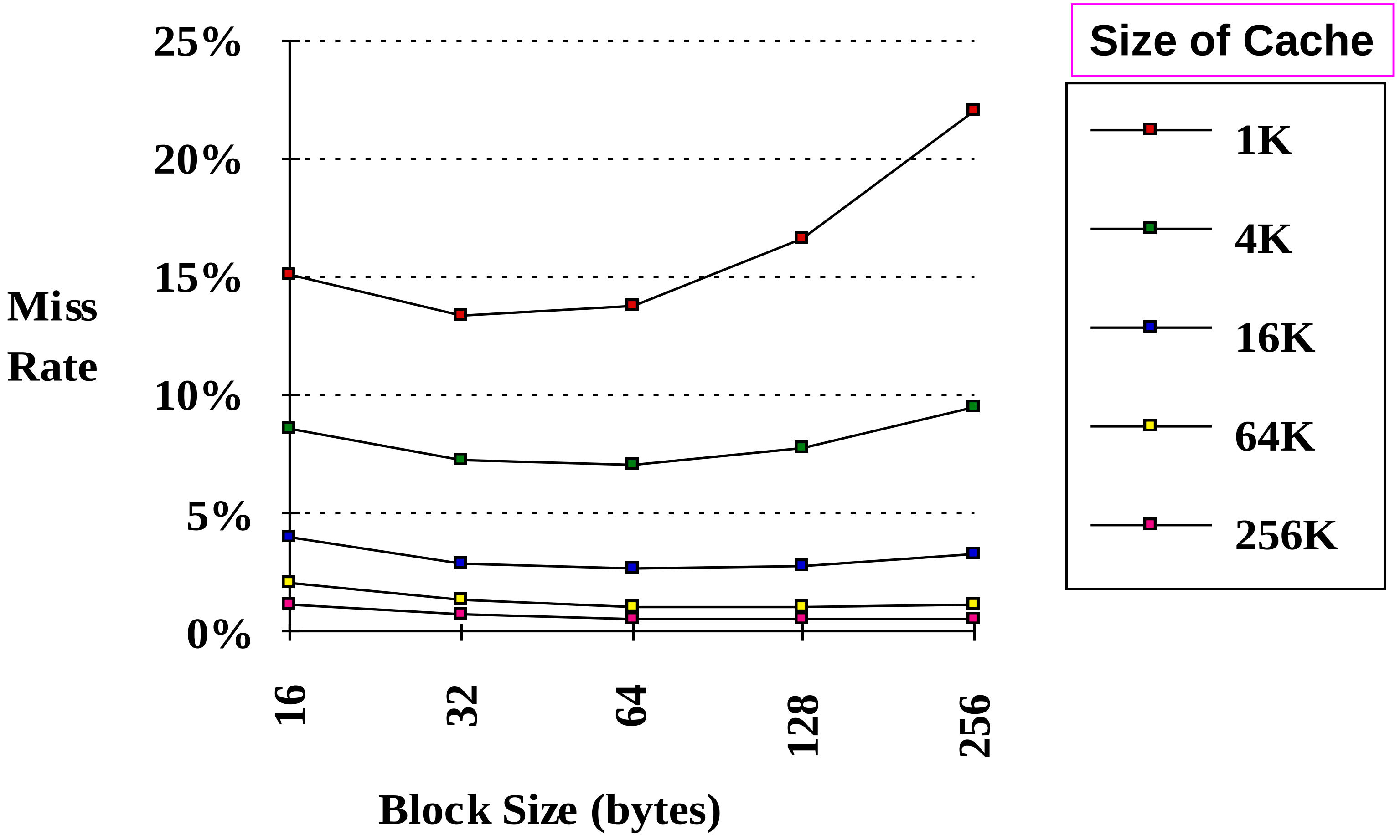
$$\boxed{\text{Average Memory Access Time} = \underset{\substack{\uparrow \\ \text{Section 5.5}}}{\text{Hit Time}} + \underset{\substack{\uparrow \\ \text{Section 5.3}}}{\text{Miss Rate}} * \underset{\substack{\uparrow \\ \text{Section 5.4}}}{\text{Miss Penalty}}}$$

$$CPUtime = IC * (CPI_{Execution} + \frac{MemoryAccess}{Instruction} * MissRate * MissPenalty * ClockCycleTime)$$

- **Improve performance by:**
  1. **Reduce the miss rate,**
  2. **Reduce the miss penalty, or**
  3. **Reduce the time to hit in the cache.**

Reducing Cache Misses: 1. Larger Block Size

Using the principle of locality. The larger the block, the greater the chance parts of it will be used again.

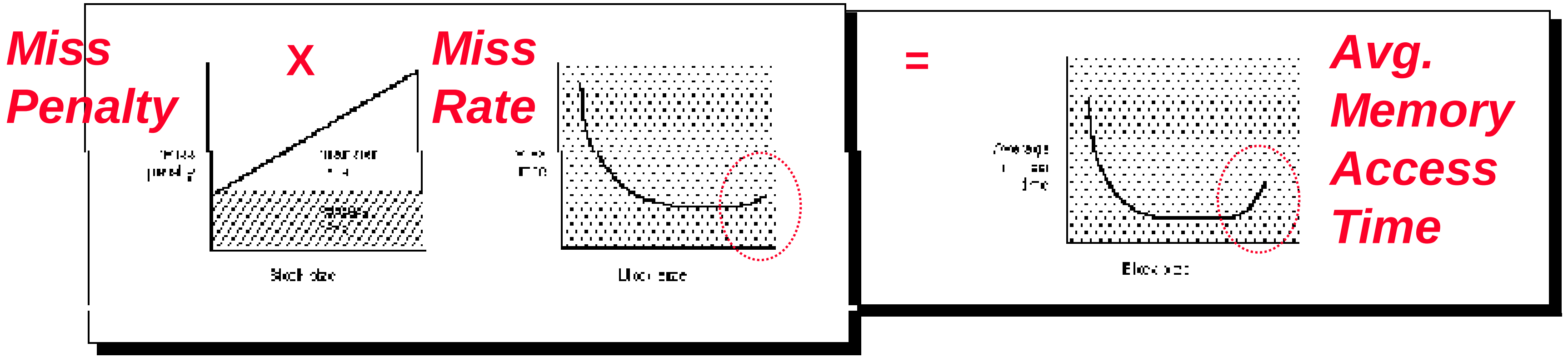


Increasing Block Size

- One way to reduce the miss rate is to increase the block size
  - Take advantage of spatial locality
  - Decreases compulsory misses
- However, larger blocks have disadvantages
  - May increase the miss penalty (need to get more data)
  - May increase hit time (need to read more data from cache and larger mux)
  - May increase miss rate, since conflict misses
- Increasing the block size can help, but don't overdo it.

Block Size vs. Cache Measures

- **Increasing Block Size generally increases Miss Penalty and decreases Miss Rate**
- **As the block size increases the AMAT starts to decrease, but eventually increases**



Block Size **Block Size** **Block Size**

Reducing Cache Misses: 2. Higher Associativity

- Increasing associativity helps reduce conflict misses
- 2:1 Cache Rule:
  - The miss rate of a direct mapped cache of size N is about equal to the miss rate of a 2-way set associative cache of size N/2
  - For example, the miss rate of a 32 Kbyte direct mapped cache is about equal to the miss rate of a 16 Kbyte 2-way set associative cache
- Disadvantages of higher associativity
  - Need to do large number of comparisons
  - Need n-to-1 multiplexor for n-way set associative
  - Could increase hit time

AMAT vs. Associativity

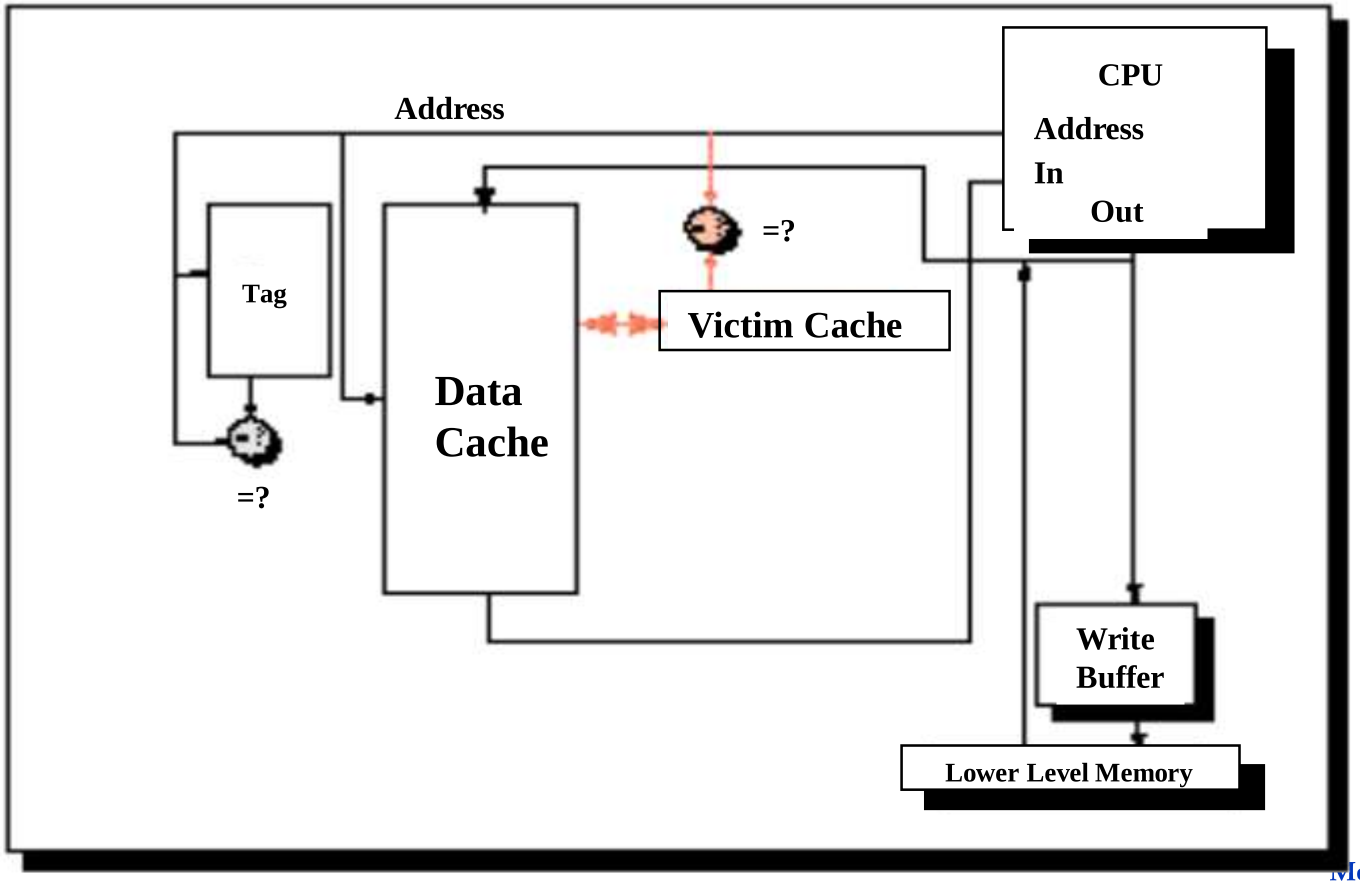
Cache Size	Associativity			
(KB)	1-way	2-way	4-way	8-way
1	7.65	6.60	6.22	5.44
2	5.90	4.90	4.62	4.09
4	4.60	3.95	3.57	3.19
8	3.30	3.00	2.87	2.59
16	2.45	2.20	2.12	2.04
32	2.00	1.80	1.77	1.79
64	1.70	1.60	1.57	1.59
128	1.50	1.45	1.42	1.44

Red means A.M.A.T. not improved by more associativity

Does not take into account effect of slower clock on rest of program

Reducing Cache Misses: 3. Victim Cache

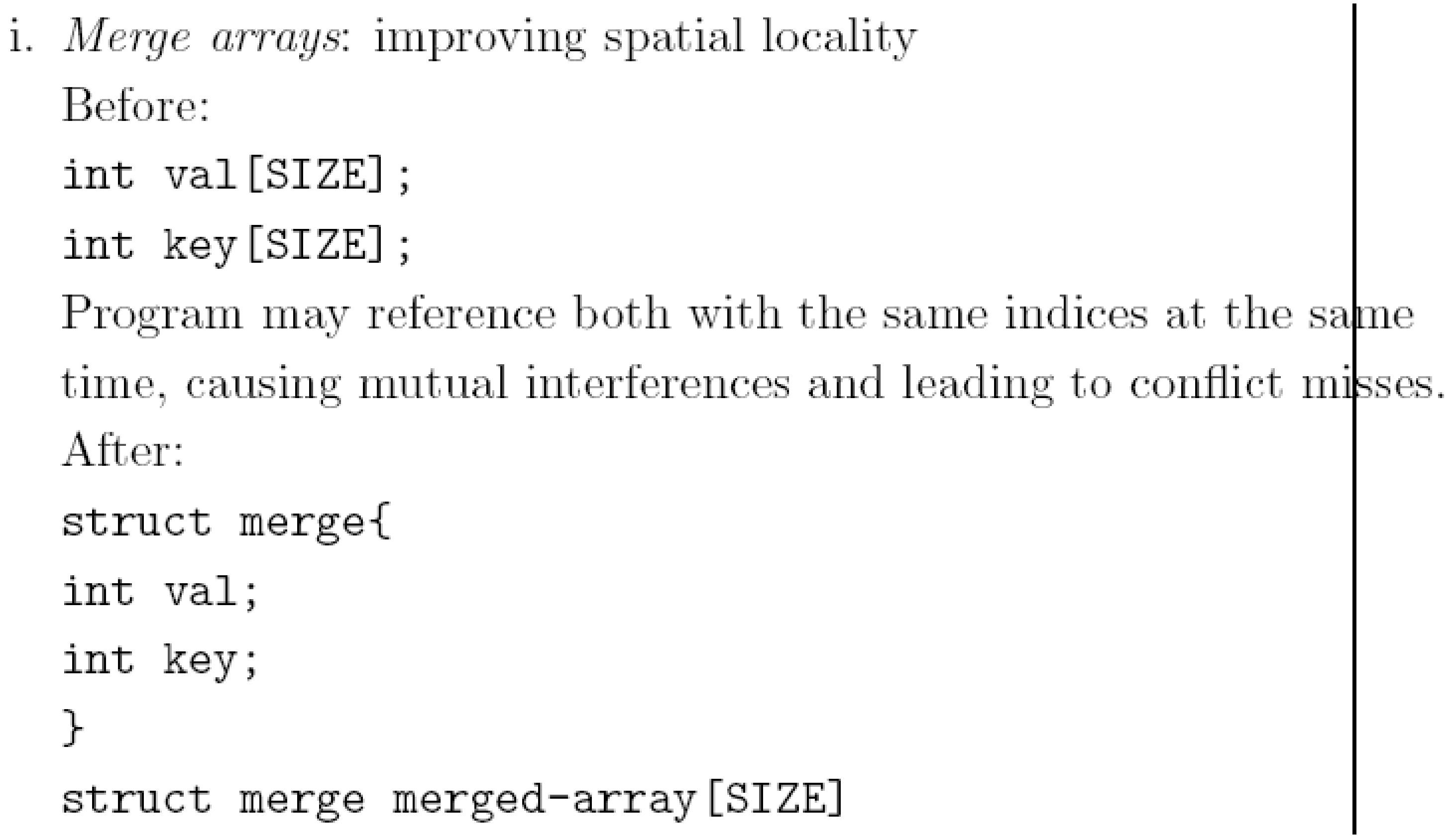
- Data discarded from cache is placed in an extra small buffer (victim cache).
- On a cache miss check victim cache for data before going to main memory
- Jouppi [1990]: A 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
- Used in Alpha, HP PA-RISC CPUs.



Reducing Cache Misses:
4. Way Prediction and Pseudoassociative Caches

- ***Way prediction*** **helps select one block among those in a set, thus requiring only one tag comparison (if hit).**
  - **Preserves advantages of direct-mapping (why?);**
  - **In case of a miss, other block(s) are checked.**
- ***Pseudoassociative*** **(also called column associative) caches**
  - **Operate exactly as direct-mapping caches when hit, thus again preserving advantages of the direct-mapping;**
  - **In case of a miss, another block is checked (as if in set-associative caches), by simply inverting the most significant bit of the index field to find the other block in the “pseudoset”.**
  - **real hit time < pseudo-hit time**
  - **too many pseudo hits would defeat the purpose**

Reducing Cache Misses:
5. Compiler Optimizations



i. *Merge arrays*: improving spatial locality

Before:

```
int val[SIZE];
int key[SIZE];
```

Program may reference both with the same indices at the same time, causing mutual interferences and leading to conflict misses.

After:

```
struct merge{
int val;
int key;
}
struct merge merged-array[SIZE]
```

Reducing Cache Misses:
5. Compiler Optimizations

ii. *Loop exchange*: improving spatial locality and maximizing the use of data already in cache before they are discarded.

Before:

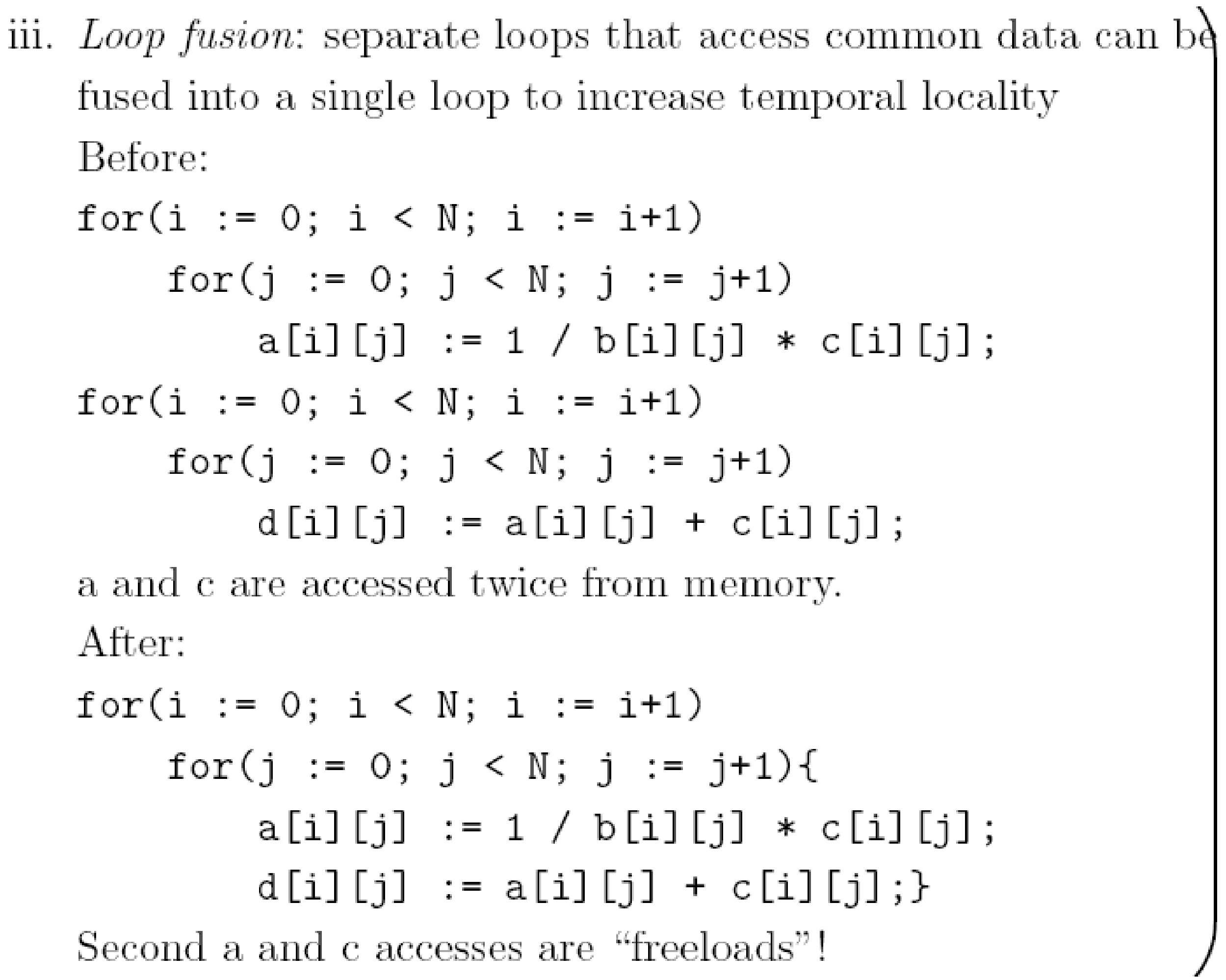
```
for(j := 0; j < 100; j := j+1)
   for(i := 0; i < 5000; i := i+1)
       x[i][j] := 2 * x[i][j]
```

Program skips through memory in strides of 100 words (consecutive column access)

After:

```
for(i := 0; i < 5000; i := i+1)
   for(j := 0; j < 100; j := j+1)
       x[i][j] := 2 * x[i][j]
```

Reducing Cache Misses:
5. Compiler Optimizations



iii. *Loop fusion*: separate loops that access common data can be fused into a single loop to increase temporal locality

Before:

```
for(i := 0; i < N; i := i+1)
    for(j := 0; j < N; j := j+1)
        a[i][j] := 1 / b[i][j] * c[i][j];
for(i := 0; i < N; i := i+1)
    for(j := 0; j < N; j := j+1)
        d[i][j] := a[i][j] + c[i][j];
```

a and c are accessed twice from memory.

After:

```
for(i := 0; i < N; i := i+1)
    for(j := 0; j < N; j := j+1){
        a[i][j] := 1 / b[i][j] * c[i][j];
        d[i][j] := a[i][j] + c[i][j];}
```

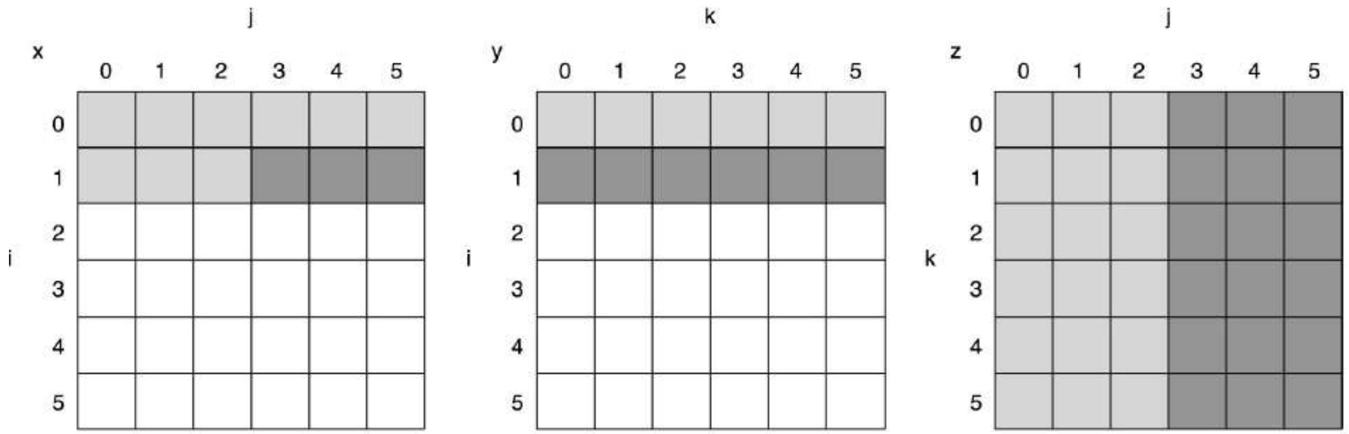
Second a and c accesses are “freeloads”!

Reducing Cache Misses:
5. Compiler Optimizations

- *Blocking*: improve temporal and spatial locality
  a) multiple arrays are accessed in both ways (i.e., row-major and column-major), namely, orthogonal accesses that can not be helped by earlier methods
  b) concentrate on submatrices, or blocks

```
for(i := 0; i < N; i := i+1)
    for(j := 0; j < N; j := j+1){
        r := 0;
        for(k := 0; k < N; k := k+1)
            r := r + y[i][k] * z[k][j];
        x[i][j] := r;}
```

  c) All *N*N* elements of Y and Z are accessed *N* times and each element of X is accessed once. Thus, there are N^3 operations and $2N^3 + N^2$ reads! Capacity misses are a function of *N* and cache size in this case.

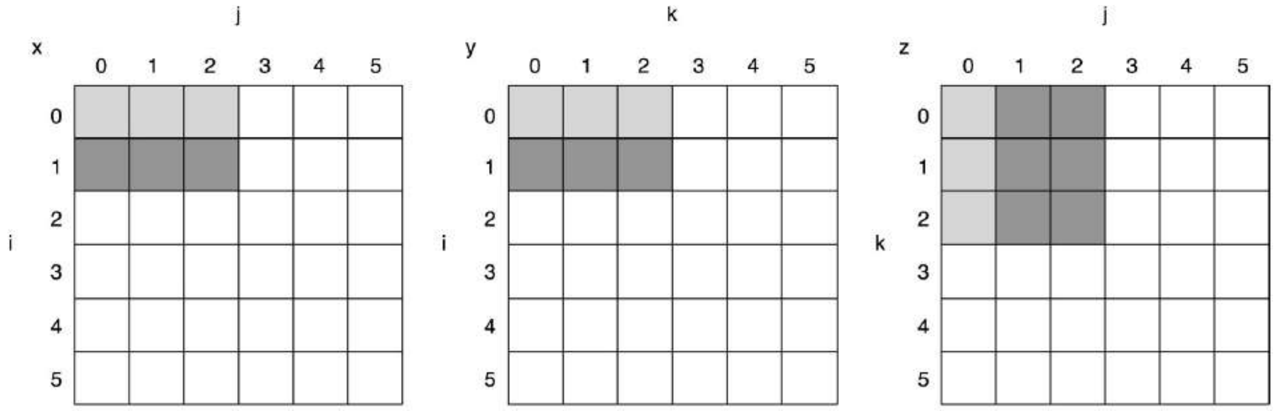


Reducing Cache Misses:
5. Compiler Optimizations

- *Blocking*: improve temporal and spatial locality
  - a) To ensure that elements being accessed can fit in the cache, the original code is changed to compute a submatrix of size *B*B,* where *B* is called the ***blocking factor***.
  - b) To total number of memory words accessed is $2N^3/B + N^2$
  - c) Blocking exploits a combination of spatial (Y) and temporal (Z) locality.

After:

```
for(jj := 0; jj < N; jj := jj+B)
for(kk := 0; kk < N; kk := kk+B)
for(i := 0; i < N; i := i+1)
    for(j := jj; j < min(jj+B-1,N); j := j+1){
        r := 0;
        for(k := kk; k < min(kk+B-1,N); k := k+1)
            r := r + y[i][k] * z[k][j];
```



Reducing Cache Miss Penalty:
1. Multi-level Cache

a) To keep up with the widening gap between CPU and main memory, try to:

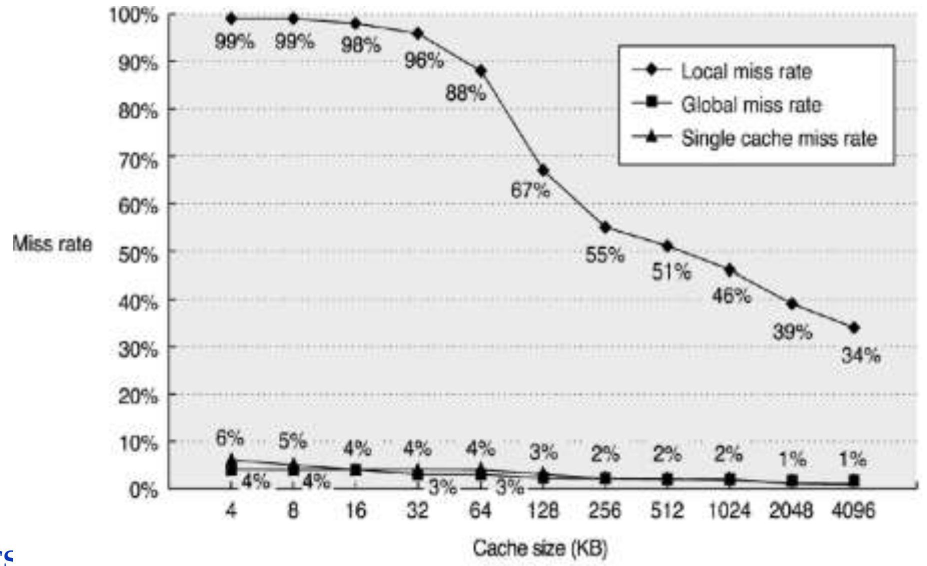
   i. make cache faster, and

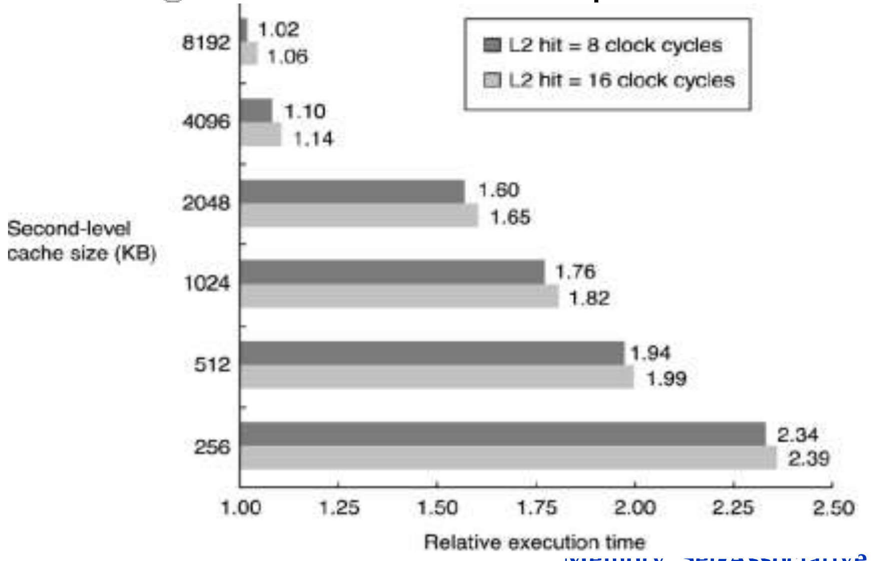
   ii. make cache larger

   by adding another, larger but slower cache between cache and the main memory.

$$AMAT = HT_{L1} + MR_{L1} \times \underbrace{(HT_{L2} + MR_{L2} \times MP_{L2})}_{MP_{L1}}$$

$$= HT_{L1} + HT_{L2} \times MR_{L1} + \underbrace{MR_{L1} \times MR_{L2}}_{\text{global miss rate}} \times MP_{L2}$$





Adding an L2 Cache

- If a direct mapped cache has a hit rate of 95%, a hit time of 4 ns, and a miss penalty of 100 ns, what is the AMAT?

- If an L2 cache is added with a hit time of 20 ns and a hit rate of 50%, what is the new AMAT?

Adding an L2 Cache

- If a direct mapped cache has a hit rate of 95%, a hit time of 4 ns, and a miss penalty of 100 ns, what is the AMAT?

  AMAT = Hit time + Miss rate x Miss penalty = 4 + 0.05 x 100 = 9 ns

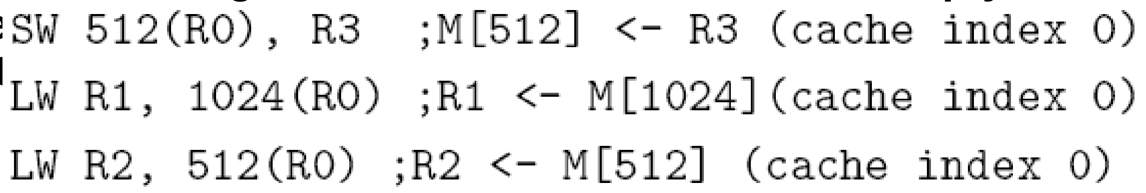
- If an L2 cache is added with a hit time of 20 ns and a hit rate of 50%, what is the new AMAT?

  AMAT = Hit Time_{L1} + Miss Rate_{L1} x (Hit Time_{L2} + Miss Rate_{L2} x Miss Penalty_{L2})

  =4 + 0.05 x (20 + 0.5x100) = 7.5 ns

Reducing Cache Miss Penalty:
2. Handling Misses Judiciously

- **Critical Word First and Early Restart**
  - CPU needs just one word of the block at a time:
    - *critical word first:* fetch the required word first, and
    - *early start:* as soon as the required word arrives, send it to CPU.
- **Giving Priority to Read Misses over Write Misses**
  - Serves reads before writes have been completed:
    - while write buffers improve write-through performance, they complicate memory accesses by potentially delaying updates to memory;
    - instead of waiting for the write buffer to become empty before proce t that might



```
SW 512(R0), R3   ;M[512] <- R3 (cache index 0)
LW R1, 1024(R0) ;R1 <- M[1024](cache index 0)
LW R2, 512(R0) ;R2 <- M[512] (cache index 0)
```

    - in a write-back scheme, the dirty copy upon replacing is first written to the write buffer instead of the memory, thus improving performance.

Reducing Cache Miss Penalty: 3. Compiler-Controlled Prefetching

- Compiler inserts prefetch instructions
- An Example

```
for(i:=0; i<3; i:=i+1)
    for(j:=0; j<100; j:=j+1)
        a[i][j] := b[j][0] * b[j+1][0]
```

  - 16-byte blocks, 8KB cache, 1-way write back, 8-byte elements; What kind of locality, if any, exists for a and b?
    a. 3 100-element rows (100 columns) visited; spatial locality: even-indexed elements miss and odd-indexed elements hit, leading to 3*100/2 = 150 misses
    b. 101 rows and 3 columns visited; no spatial locality, but there is temporal locality: same element is used in *i*th and (i + 1)st iterations and the same element is access in each *i* iteration (outer loop). 100 misses for i = 0 and 1 miss for j = 0 for a total of 101 misses
  - Assuming large penalty (50 cycles and at least 7 iterations must be prefetched). Splitting the loop into two, we have

Reducing Cache Miss Penalty:
3. Compiler-Controlled Prefetching

❖ An Example (continued)

```
for(j:=0; j<100; j:=j+1){
        prefetch(b[j+7][0];
        prefetch(a[0][j+7];
        a[0][j] := b[j][0] * b[j+1][0];};
for(i:=1; i<3; i:=i+1)
        for(j:=0; j<100; j:=j+1){
                prefetch(a[i][j+7];
                a[i][j] := b[j][0] * b[j+1][0]}
```

➢ Assuming that each iteration of the pre-split loop consumes 7 cycles and no conflict and capacity misses, then it consumes a total of 7*300 + 251*50 = 14650 cycles (total iteration cycles plus total cache miss cycles);

Reducing Cache Miss Penalty:
3. Compiler-Controlled Prefetching

- **An Example (continued)**
  - **the first loop consumes 9 cycles per iteration (due to the two prefetch instruction)**
  - **the second loop consumes 8 cycles per iteration (due to the single prefetch instruction),**
  - **during the first 7 iterations of the first loop array** a **incurs 4 cache misses,**
  - **array** b **incurs 7 cache misses,**
  - **during the first 7 iterations of the second loop for *i = 1* and *i = 2* array** a **incurs 4 cache misses each**
  - **array** b **does not incur any cache miss in the second split!.**
  - **the split loop consumes a total of (1+1+7)*100+(4+7)*50+(1+7)*200+(4+4)*50 = 3450**
  - **Prefetching improves performance: 14650/3450=4.25 folds**

Reducing Cache Hit Time:

- **Small and simple caches**
  - ***smaller is faster:***
    - **small index, less address translation time**
    - **small cache can fit on the same chip with CPU**
    - **low associativity: in addition to a simpler/shorter tag check, 1-way cache allows overlapping tag check with transmission of data which is not possible with any higher associativity!**
- **Avoid address translation during indexing**
  - ***Make the common case fast:***
    - **use virtual address for cache because most memory accesses (more than 90%) take place in cache, resulting in virtual cache**

Reducing Cache Hit Time:

- ***Make the common case fast (continued):***
  - **there are at least three important performance aspects that directly relate to virtual-to-physical translation:**
    1) **improperly organized or insufficiently sized TLBs may create excess not-in-TLB faults, adding time to program execution time**
    2) **for a physical cache, the TLB access time must occur before the cache access, extending the cache access time**
    3) **two-line address (e.g., an I-line and a D-line address) may be independent of each other in virtual address space yet collide in the real address space, when they draw pages whose lower page address bits (and upper cache address bits) are identical**
  - **problems with virtual cache:**
    1) **Page-level protection must be enforced no matter what during address translation (solution: copy protection info from TLB on a miss and hold it in a field for future virtual indexing/tagging)**
    2) **when a process is switched in/out, the entire cache has to be flushed out 'cause physical address will be different each time, i.e., the problem of context switching (solution: *process identifier tag -- PID*)**

Reducing Cache Hit Time:

- **Avoid address translation during indexing** (continued)
  - **problems with virtual cache:**
    - **3) different virtual addresses may refer to the same physical address, i.e., the problem of synonyms/aliases**
      - ✓ ***HW solution: guarantee every cache block a unique phy. Address***
      - ✓ ***SW solution: force aliases to share some address bits (e.g., page-coloring)***
      - ✓ ***Virtually indexed and physically tagged***
- **Pipelined cache writes**
  - ❖ ***the solution is to reduce CCT and increase # of stages – increases instr. throughput***
- **Trace caches**
  - ❖ ***Finds a dynamic sequence of instructions including taken branches to load into a cache block:***
    - **Put traces of the executed instructions into cache blocks as determined by the CPU**
    - **Branch prediction is folded in to the cache and must be validated along with the addresses to have a valid fetch.**
    - **Disadvantage: store the same instructions multiple times**

Cache Performance Measures

- ***Hit rate***: fraction found in the cache
  - So high that we usually talk about ***Miss rate = 1 - Hit Rate***
- ***Hit time***: time to access the cache
- ***Miss penalty***: time to replace a block from lower level, including time to replace in CPU
  - ***access time***: time to access lower level
  - ***transfer time***: time to transfer block
- **Average memory-access time (AMAT)**

  = Hit time + Miss rate x Miss penalty (ns or clocks)

Cache performance

- **Miss-oriented Approach to Memory Access:**

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty \right) \times CycleTime$$

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemMisses}{Inst} \times MissPenalty \right) \times CycleTime$$

  - $\text{CPI}_{\text{Execution}}$ includes ALU and Memory instructions

- **Separating out Memory component entirely**
  - AMAT = Average Memory Access Time
  - $\text{CPI}_{\text{ALUOps}}$ does not include memory instructions

$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

$$\begin{aligned} AMAT &= HitTime + MissRate \times MissPenalty \\ &= \left(HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst} \right) + \\ &\quad \left(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data} \right) \end{aligned}$$

Calculating AMAT

- If a direct mapped cache has a hit rate of 95%, a hit time of 4 ns, and a miss penalty of 100 ns, what is the AMAT?

- If replacing the cache with a 2-way set associative increases the hit rate to 97%, but increases the hit time to 5 ns, what is the new AMAT?

Calculating AMAT

- If a direct mapped cache has a hit rate of 95%, a hit time of 4 ns, and a miss penalty of 100 ns, what is the AMAT?

  AMAT = Hit time + Miss rate x Miss penalty = 4 + 0.05 x 100 = 9 ns

- If replacing the cache with a 2-way set associative increases the hit rate to 97%, but increases the hit time to 5 ns, what is the new AMAT?

  AMAT = Hit time + Miss rate x Miss penalty = 5 + 0.03 x 100 = 8 ns

Impact on Performance

- **Suppose a processor executes at**
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- **Suppose that 10% of data memory operations get 50 cycle miss penalty**
- **Suppose that 1% of instructions get same miss penalty**
- **Calculate AMAT?**

Impact on Performance

- **Suppose a processor executes at**
  - **Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1**
  - **50% arith/logic, 30% ld/st, 20% control**
- **Suppose that 10% of data memory operations get 50 cycle miss penalty**
- **Suppose that 1% of instructions get same miss penalty**

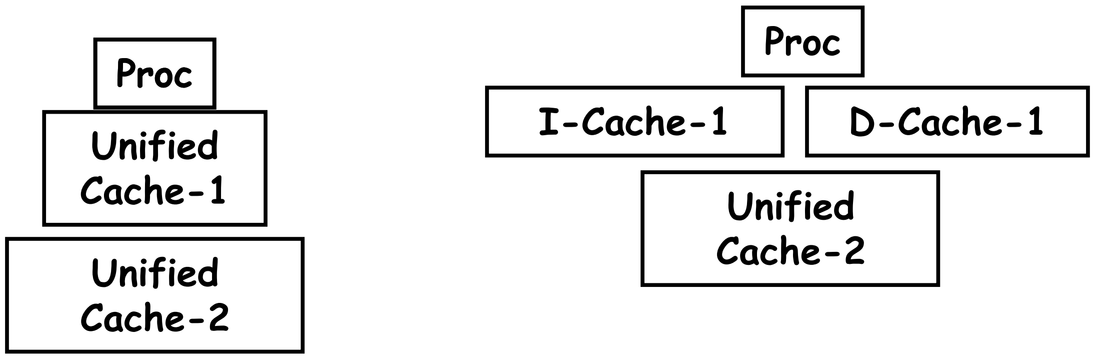
- **CPI = ideal CPI + average stalls per instruction**
  1.1(cycles/ins) +
  [0.30 (DataMops/ins) x 0.10 (miss/DataMop) x 50 (cycle/miss)] +
  [1 (InstMop/ins) x 0.01 (miss/InstMop) x 50 (cycle/miss)]
  = (1.1 + 1.5 + .5) cycle/ins = 3.1

- **AMAT=(1/1.3)x[1+0.01x50]+(0.3/1.3)x[1+0.1x50]=2.54**

$$\begin{aligned} AMAT &= HitTime + MissRate \times MissPenalty \\ &= \left(HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}\right) + \\ &\quad \left(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data}\right) \end{aligned}$$

Unified vs Split Caches

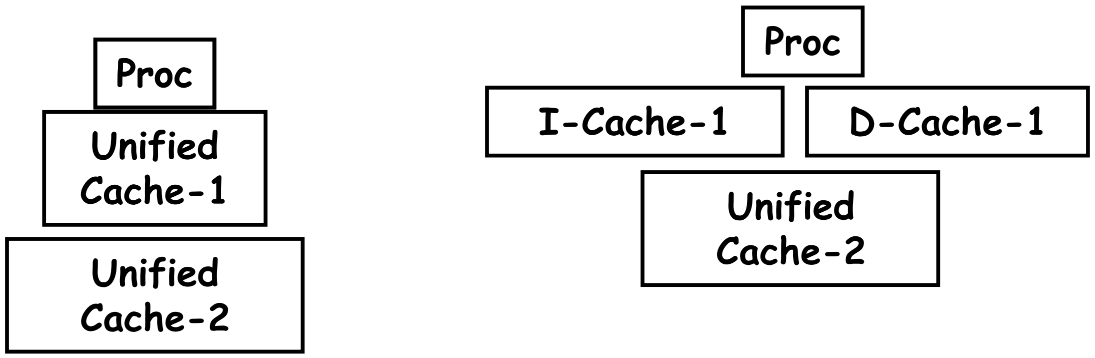
- **Unified vs Separate I&D**



- **Example:**
  - **16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%**
  - **32KB unified: Aggregate miss rate=1.99%**
- **Which is better (ignore L2 cache)?**
  - **Assume 33% data ops ⇒ 75% accesses from instructions (1.0/1.33)**
  - **hit time=1, miss time=50**
  - **Note that *data* hit has 1 stall for unified cache (only one port)**

Unified vs Split Caches

- **Unified vs Separate I&D**



- **Example:**
  - 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
  - 32KB unified: Aggregate miss rate=1.99%
- **Which is better (ignore L2 cache)?**
  - Assume 33% data ops $\Rightarrow$ 75% accesses from instructions (1.0/1.33)
  - hit time=1, miss time=50
  - Note that *data* hit has 1 stall for unified cache (only one port)

$AMAT_{Harvard}$=75%x(1+0.64%x50)+25%x(1+6.47%x50) = 2.05

$AMAT_{Unified}$=75%x(1+1.99%x50)+25%x(1+1+1.99%x50)= 2.24

Cache Performance Summary

- AMAT = Hit time + Miss rate x Miss penalty
- Split vs. Unified Cache
- 3C's of misses
  - compulsory
  - capacity
  - conflict
- Methods for improving performance
  - Reduce miss rate: increase cache size, block size, associativity, compiler optimization, way-prediction, victim cache, etc.
  - Reduce miss penalty: multi-level cache, handling misses judiciously, compiler-controlled prefetching, etc.
  - Reduce hit time: smaller and simpler caches, avoiding address translation in indexing, pipelining cache writes, trace cache, etc.