CHAPTER 5

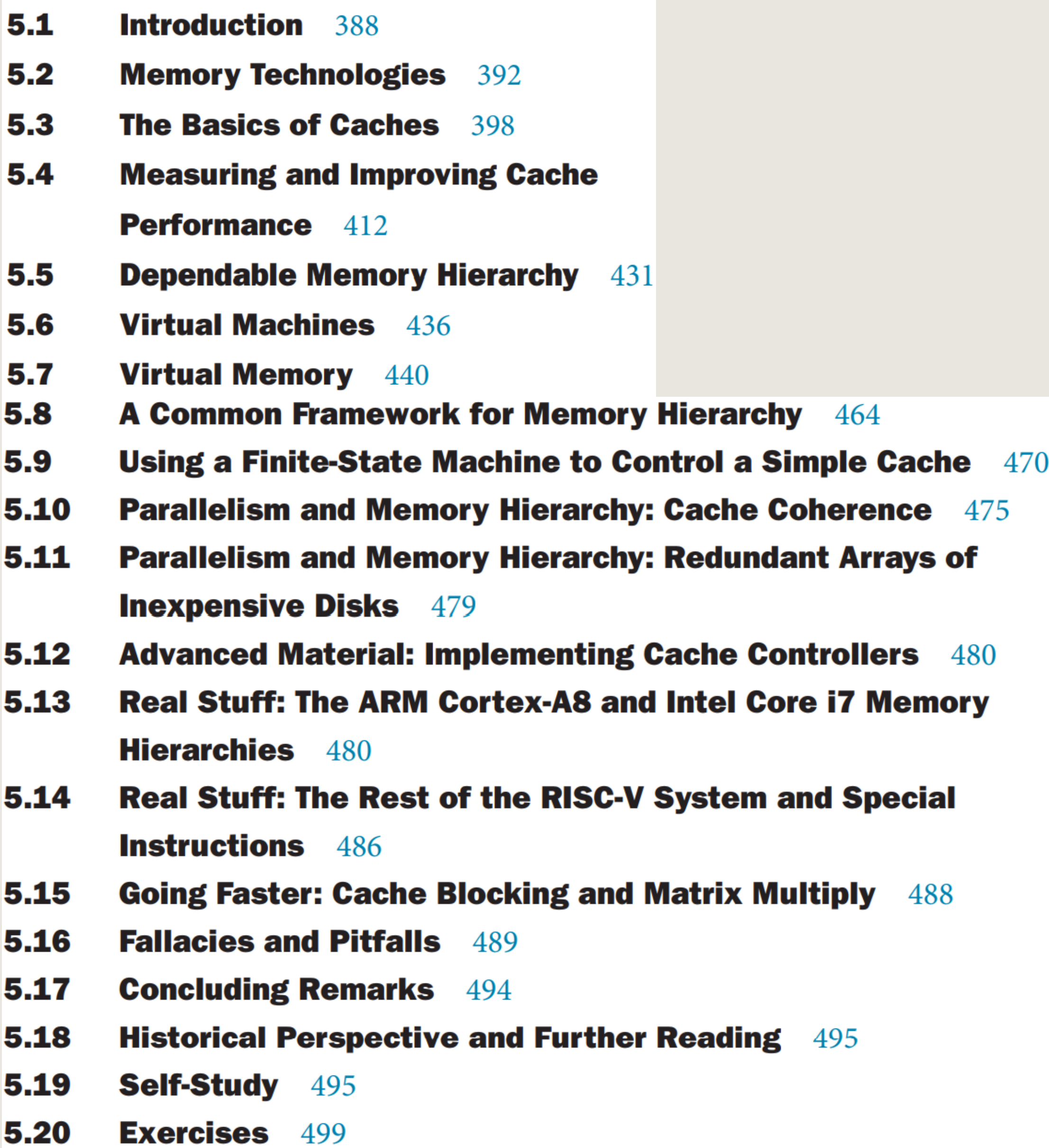
Large and Fast: Exploiting Memory Hierarchy

Chia-Heng Tu

Dept. of Computer Science and Information Engineering

National Cheng Kung University

Outline



5.1 Introduction 388
5.2 Memory Technologies 392
5.3 The Basics of Caches 398
5.4 Measuring and Improving Cache Performance 412
× 5.5 Dependable Memory Hierarchy 431
5.6 Virtual Machines 436
5.7 Virtual Memory 440
5.8 A Common Framework for Memory Hierarchy 464
× 5.9 Using a Finite-State Machine to Control a Simple Cache 470
× 5.10 Parallelism and Memory Hierarchy: Cache Coherence 475
× 5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks 479
× 5.12 Advanced Material: Implementing Cache Controllers 480
× 5.13 Real Stuff: The ARM Cortex-A8 and Intel Core i7 Memory Hierarchies 480
× 5.14 Real Stuff: The Rest of the RISC-V System and Special Instructions 486
× 5.15 Going Faster: Cache Blocking and Matrix Multiply 488
× 5.16 Fallacies and Pitfalls 489
5.17 Concluding Remarks 494
× 5.18 Historical Perspective and Further Reading 495
× 5.19 Self-Study 495
× 5.20 Exercises 499

× means not covered in this file

Chapter Goals

- From the earliest days of computing, programmers have wanted unlimited amounts of fast memory
- The topics in this chapter aid programmers by creating that illusion

- Memory systems are critical to performance,
  - ➢computer designers devote a great deal of attention to these systems and develop sophisticated mechanisms for improving the performance of the memory system
- This chapter discusses the major conceptual ideas
  - ➢although we use many simplifications and abstractions to keep the material manageable in length and complexity

Principle of Locality

- The principle of locality states that
  - programs access a relatively small portion of their address space at any instant of time
    (just as you accessed a very small portion of the school library's collection)

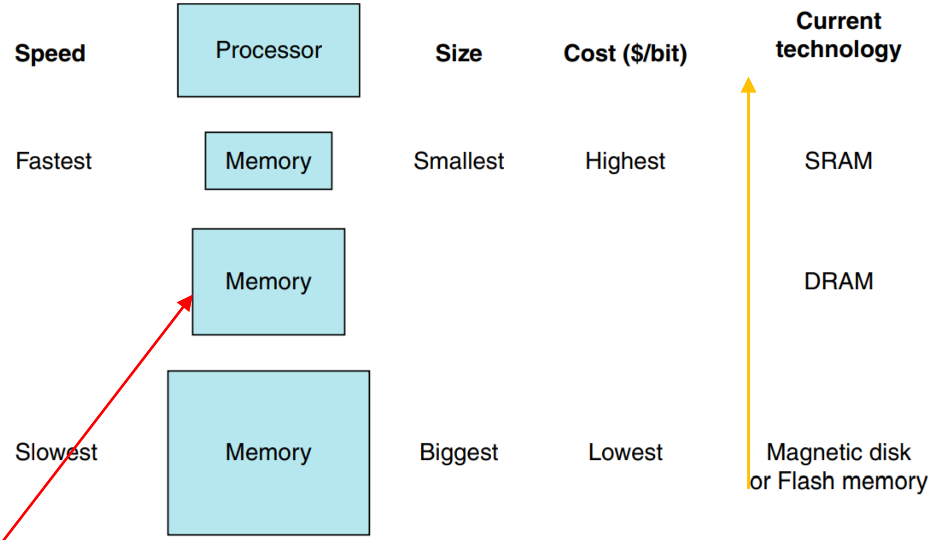
Two different types of locality:

- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, the loop index variable, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, nearby elements of a data array

Locality and Memory Hierarchy

- Computer designers take advantage of the principle of locality
  - ➢by implementing the memory of a computer as a memory hierarchy

- A memory hierarchy consists of multiple levels of memory
  - ➢with different speeds and sizes
  - ➢The faster memories are more expensive per bit than the slower memories and thus are smaller
- Data arrangement
  - ➢Store everything on disk
  - ➢Copy recently accessed (and nearby) items from disk to smaller DRAM memory (Main memory)
  - ➢Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - ➢Cache memory attached to CPU



Speed	Processor	Size	Cost ($/bit)	Current technology
Fastest	Memory	Smallest	Highest	SRAM
	Memory			DRAM
Slowest	Memory	Biggest	Lowest	Magnetic disk or Flash memory

FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many personal mobile devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 5.2

A level closer to the processor is generally a subset of any level further away

Memory Hierarchy Levels

- A data block or line
  - shown in Fig. 5.2 is the minimum unit of information
  - that can be either present or not present in the two-level hierarchy is called a block or a line
  - Is the basic unit of the copying (between two adjacent levels of memory)

- Hit: access satisfied by upper level
  - If accessed data is present in upper level
  - Hit ratio: #hits/#accesses
  - Hit time is the time to access the upper level of the memory hierarchy, including to check the "hit" or "miss"
- Miss: block copied from lower level
  - If accessed data is absent
  - Miss ratio: #misses/#accesses = 1 – hit ratio
  - The lower level in the hierarchy is then accessed to *retrieve* the block containing the requested data
  - Miss penalty is the time taken for the *retrieval*
  - Then, accessed data supplied from upper level
- *Hit* time is far less than *miss* penalty
  - Improve hit rate

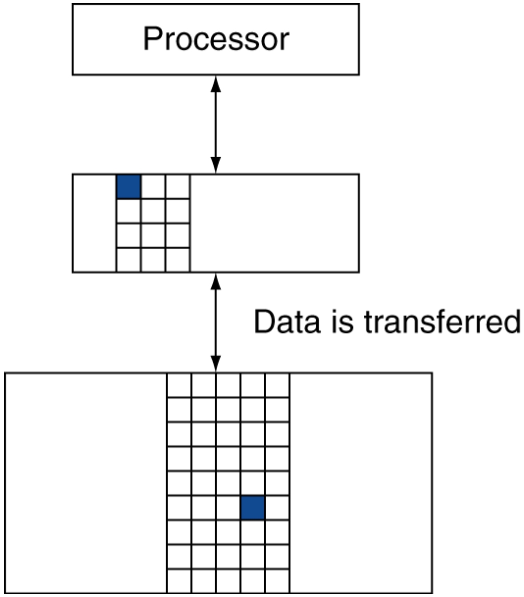


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a block or a line. Usually we transfer an entire block when we copy something between levels

More about Memory Hierarchy

- Fig. 5.3 shows that a memory hierarchy uses smaller and faster memory technologies close to the processor
  - ➢Accesses that hit in the highest level of the hierarchy can be processed quickly
  - ➢Accesses that miss go to lower levels of the hierarchy, which are larger but slower
- If the hit rate is high enough,
  - ➢the memory hierarchy has an effective *access time* close to that of the highest (and fastest) level and
  - ➢a *size* equal to that of the lowest (and largest) level
- In most systems, the memory is a true hierarchy,
  - ➢meaning that data cannot be present in level i unless they are also present in level i + 1

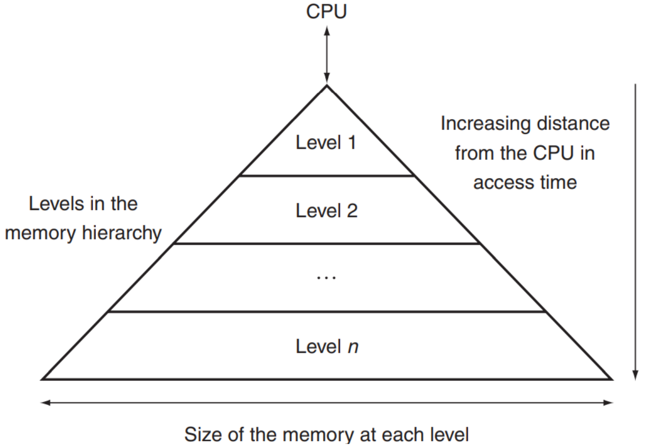

CPU
Level 1
Level 2
…
Level *n*
Increasing distance from the CPU in access time
Levels in the memory hierarchy
Size of the memory at each level

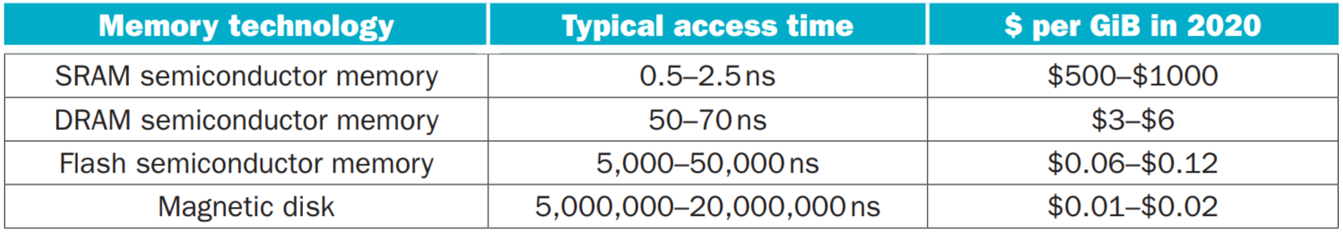
FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size. This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level n. Maintaining this illusion is the subject of this chapter. Although local storage is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy

§5.2 Memory Technologies

Memory Technology

- Four primary technologies used today in memory hierarchies
  - ➢Main memory is implemented from DRAM technology
  - ➢Processor caches are implemented with SRAM
  - ➢Third technology is flash memory and this nonvolatile memory is the secondary memory in Personal Mobile Devices
  - ➢Magnetic disk is used to implement the largest and slowest level in the hierarchy in servers

- Ideal memory
  - ➢Access time of SRAM
  - ➢Capacity and cost/GB of disk



Memory technology	Typical access time	$ per GiB in 2020
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$3–$6
Flash semiconductor memory	5,000–50,000 ns	$0.06–$0.12
Magnetic disk	5,000,000–20,000,000 ns	$0.01–$0.02

Please read through…

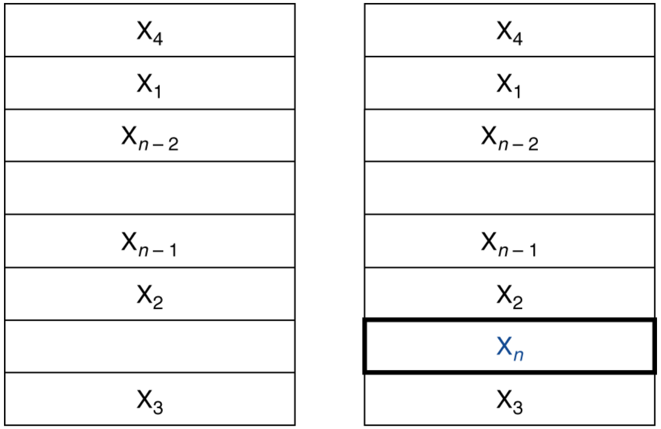
P. 393~398

For SRAM, DRAR, Flash, and Disk Memory

Cache Memory

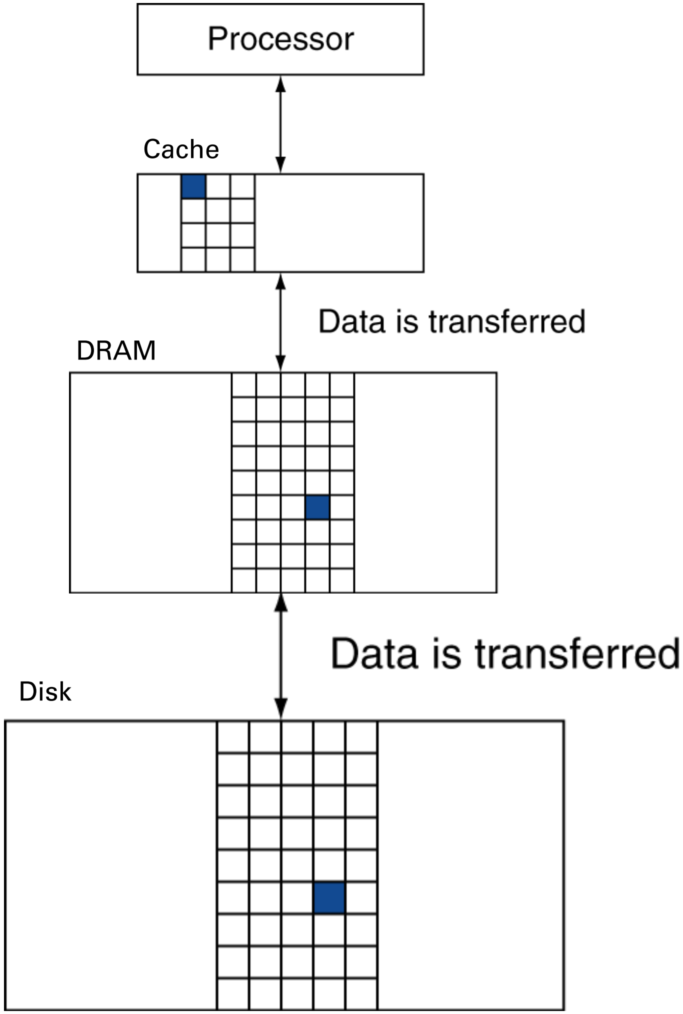
§5.3 The Basics of Caches

- Cache memory
  - The level of the memory hierarchy closest to the CPU
  - The memories in the datapath in Ch. 4 are simply replaced by caches
  - Today, *cache* is also used to refer to any storage managed to take advantage of locality of access
- Given the data access sequence: X_1, ⋯, X_{n-1}, X_n
  - Before the request, the cache contains a collection of recent references X_1 , X_2 , ⋯, X_{n-1} (as in Fig. 5.7a), and
  - the processor requests a word X_n that is not in the cache
  - This request results in a miss, and the word X_n is brought from memory into the cache
  - Assume a simple cache in which the processor requests are each one word
- The above scenario raises two questions
  - How do we know if the data is present in cache? If so, how to find it?
  - Where do we look?



a. Before the reference to X_n b. After the reference to X_n

FIGURE 5.7 The cache just before and just after a reference to a word X_n that is not initially in the cache. This reference causes a miss that forces the cache to fetch X_n from memory and insert it into the cache



Cache Design
1: Direct Mapped

- If each word can go in exactly one place in cache,
  - then it is straightforward to find the word if it is in the cache
- Direct Mapped Cache
  - Addresses → cache locations
  - The simplest way to assign the cache location based on the address of the word in memory
  - Each memory location is mapped directly to exactly one location in the cache
  - A typical mapping method:

Block Address modulo
Number of Blocks in the Cache

  - Use low-order address bits

#Blocks (=8) is a power of 2 and the modulo can be computed simply by using the low-order log2 (cache size in blocks $\log_2(8)=3$) bits of the address

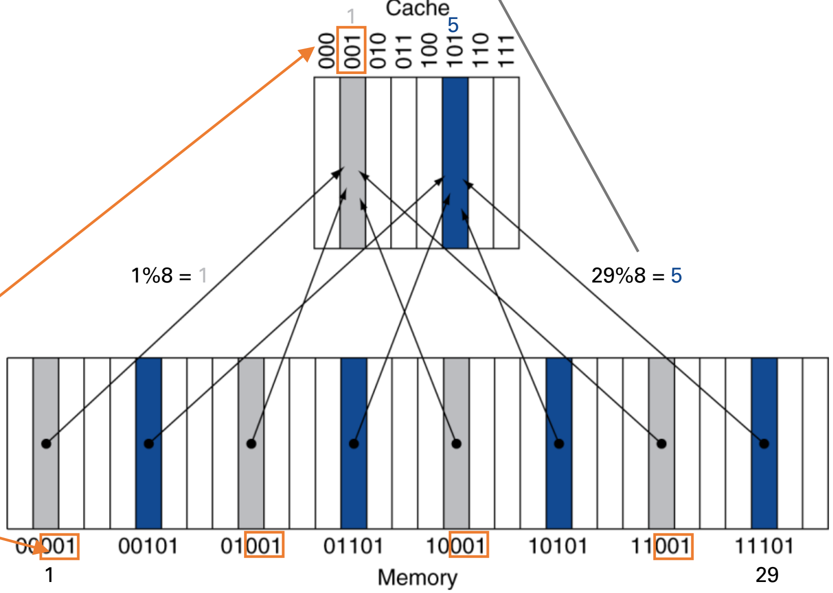
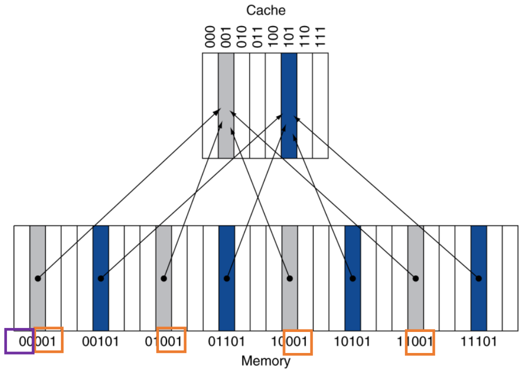


FIGURE 5.8 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the direct-mapped cache word X modulo 8. That is, the low-order log2 (8) = 3 bits are used as the cache index. Thus, addresses 00001_{two}, 01001_{two}, 10001_{two}, and 11001_{two} all map to entry 001_{two} of the cache, while addresses 00101_{two}, 01101_{two}, 10101_{two}, and 11101_{two} all map to entry 101_{two} of the cache

Tags and Valid Bits in a Cache

- Each cache location can contain the contents of a number of different memory locations
  - ➢ But, how do we know whether the data in the cache corresponds to a requested word?
  - → Answer is to add a set of tags to the cache
  - ➢ The tags contain the address information required to identify whether a word in the cache corresponds to the requested word
  - ➢ The tag needs just to contain the upper portion of the address, corresponding to the bits that are not used as an index into the cache
  - ➢ The upper two of the five address bits in the tag, since the lower 3-bit index field of the address selects the block

- We also need a way to recognize that a cache block does not have valid information
  - ➢ E.g., when a processor starts up, the cache does not have good data, and the tag fields will be meaningless
  - ➢ These tags should be ignored in such a case
  - ➢ Add a valid bit to indicate whether an entry contains a valid address (1)
  - ➢ If the bit is not set (0), there cannot be a match for this block



→ Each cache location contains data and metadata (e.g., tag and valid bit)

Tag: A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word

Valid bit: A field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains valid data

Example: Cache Behaviors (hit/miss)

- Given a sequence of nine memory references (addresses)
  - 10110, 11010, 10110, 11010, 10000, 00011,10000, 10010, 10000
- The references are issued to an empty eight-block cache
  - The low-order 3 bits of an address give the block number (index)
- The organization of the cache is shown below

Index to cache block

Valid bit (Yes:1, No:0)

Content of a memory block

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Example: Cache Behaviors

Sequence of mem ref.

Result of each access

Ref. to index (of cache block)

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two} 1	miss (5.9b)	$(10110_{two}$ mod 8) = 110_{two} 2
26	11010_{two}	miss (5.9c)	$(11010_{two}$ mod 8) = 010_{two}
22	10110_{two}	hit	$(10110_{two}$ mod 8) = 110_{two}
26	11010_{two}	hit	$(11010_{two}$ mod 8) = 010_{two}
16	10000_{two}	miss (5.9d)	$(10000_{two}$ mod 8) = 000_{two}
3	00011_{two}	miss (5.9e)	$(00011_{two}$ mod 8) = 011_{two}
16	10000_{two}	hit	$(10000_{two}$ mod 8) = 000_{two}
18	10010_{two}	miss (5.9f)	$(10010_{two}$ mod 8) = 010_{two}
16	10000_{two}	hit	$(10000_{two}$ mod 8) = 000_{two}

- For each memory reference, we do the following:
  1. Obtain memory address (10 110)
  2. Calculate the cache block index (110)
  3. Hit/Miss in cache (miss; since 110 in Fig. 5.9a is invalid)
  4. If miss, retrieve the content from the lower-level memory (Or, replace the current content)
- On the eighth reference (10 010)
  - ➢ we have conflicting demands for a block (11 010 is already in 010 as in Fig. 5.9e),
  - ➢ but the word at address 18 (10010_{two}) should be brought into cache block 2 (010_{two})
  - ➢ Hence, it must replace the word at mem. address 26 (11010_{two}), which is already in cache block 2 (010_{two})

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110 3	Y	10_{two}	4 Memory (10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

FIGURE 5.9 The cache contents are shown after each reference request that misses, with the index and tag fields shown in binary for the sequence of addresses on page 402. The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), 10010_{two} (miss), and 10000_{two} (hit). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss. The tag field will contain only the upper portion of the address. The full address of a word contained in cache block i with tag field j for this cache is j×8+ i, or equivalently the concatenation of the tag field j and the index i. For example, in cache f above, index 010_{two} has tag 10_{two} and corresponds to address 10010_{two}

Cache Organization

- The example cache has
  - ➢1,024 blocks (2^2 = 4 bytes for each word)
  - ➢Offset 2 bits (offset of between *words*) ← Aligned accesses are assumed
- Cache block index is 10 bits (2^{10} = 1,024 blocks)
  - ➢Cache tag is the upper 20 bits (=32-10-2)
  - ➢Valid bit (initialized to 0)
- A cache block address = Tag + Index
  - ➢The block address is matched with the given memory address
  - ➢See the image for illustration

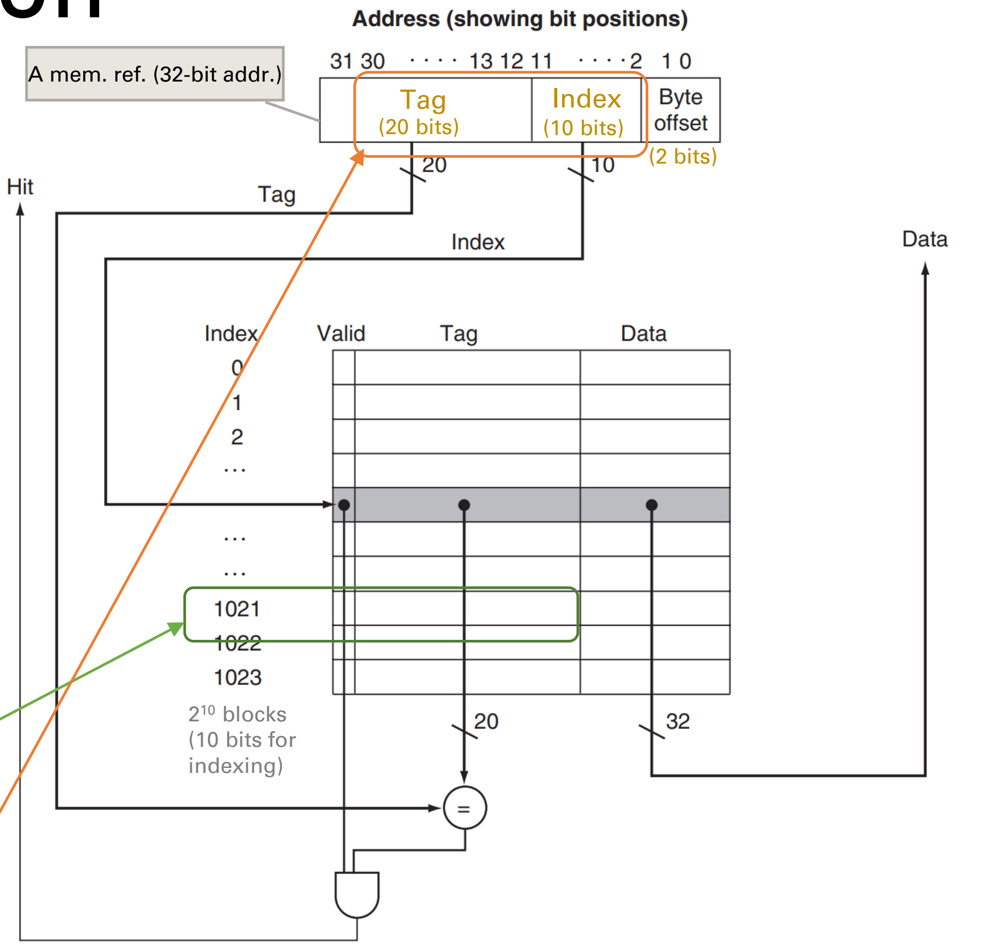


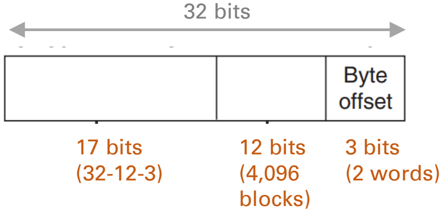
FIGURE 5.10 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. This cache holds 1024 words or 4 KiB. Unless noted otherwise, we assume 32-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 210 (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving 32 − 10 − 2 = 20 bits to be compared against the tag. If the tag and upper 52 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs

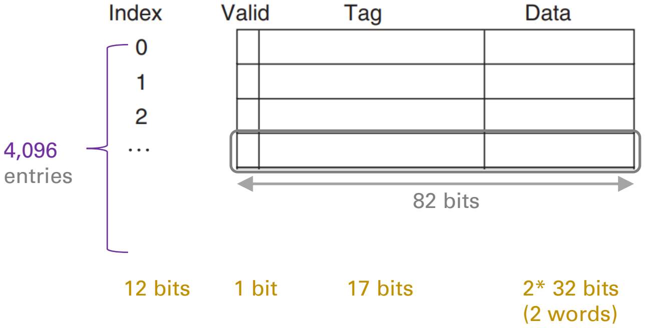
Example: Design a Cache

- The total number of bits needed for a cache
  - is a function of the cache size and the address size,
- A direct-mapped cache has
  - 32-bit addresses
  - Cache size: 4,096 blocks
    - *n* bits are used for the index
  - Cache block size: 2 words

- What is
  - the bit width for byte offset: 3
    - 2 words = 8 bytes = 2^{3}
  - the bit width of *index*: 12
    - 4,096 blocks = 2^{12} blocks
  - the bit width for tag: 17
    - 32 – 3 – 12 = 17
  - the cache entry size (bits): 82
    - 1 (valid bit) + 17 (tag) + 32*2 (data)
  - the cache size (bits): 335,872
    - 4,096 (cache entries) * [1 (valid bit) + 17 (tag) + 32*2 (data)]

  - You can check another example in p. 404 of the textbook





Large or Small Cache Block Sizes

- Larger blocks have lower miss rates
  - ➢thanks to spatial locality

- Increasing the block size will lead to two problems (for a fixed-size cache):

1. The miss rate may go up eventually
   - ➢if the block size becomes a significant fraction of the cache size,
   - ➢because the number of blocks that can be held in the cache will become small, &
   - ➢there will be a great deal of competition for those blocks

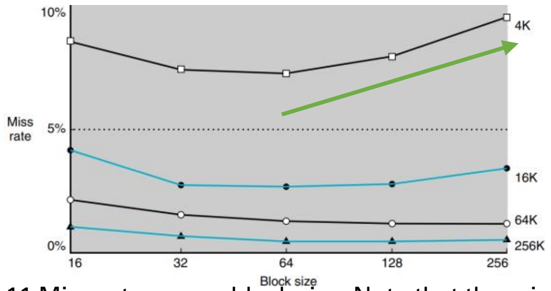
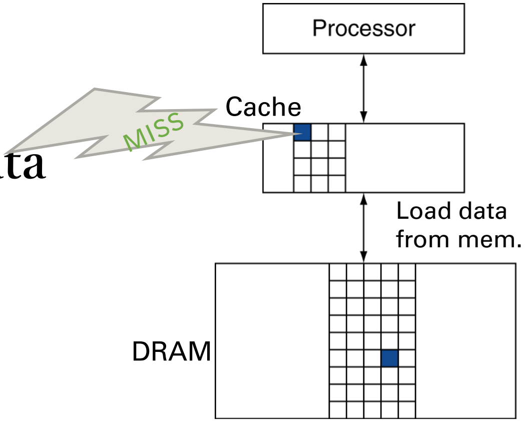


FIGURE 5.11 Miss rate versus block size. Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.)

2. The cost of a miss rises
   - ➢ The improvement in the miss rate starts to decrease as the blocks become larger (and have larger miss penalty)
   - ➢ The miss penalty is determined by the time required to [fetch the block from the next lower level of the hierarchy]() and [load it into the cache]()
   - ➢ The time to fetch the block has two parts
     - ❖the latency to the first word and
     - ❖the transfer time for the rest of the block
       → reduce the transfer time can alleviate the miss penalty
       → One opt. technique: *early restart* is to resume execution as soon as the requested word of the block is returned, rather than wait for the entire block

Handling Cache Misses

- Cache miss
  - ➢A request for data from the cache that cannot be filled because the data are not present in the cache
  - ➢I.e., inst. cache and data cache



- On cache hit, CPU proceeds normally
- On cache miss
  - ➢Stall the CPU pipeline
  - ➢Fetch block from next level of hierarchy

- Instruction cache miss
  - ➢Restart instruction fetch
- Data cache miss
  - ➢Complete data access

Steps taken on an **instruction cache** miss:

1. Send the original PC value to the memory
2. Instruct main memory to perform a read and wait for the memory to complete its access
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on if it was not on already
4. Restart the instruction execution at the first step, which will refetch the instruction, this time finding it in the cache

The steps are similar to data cache misses

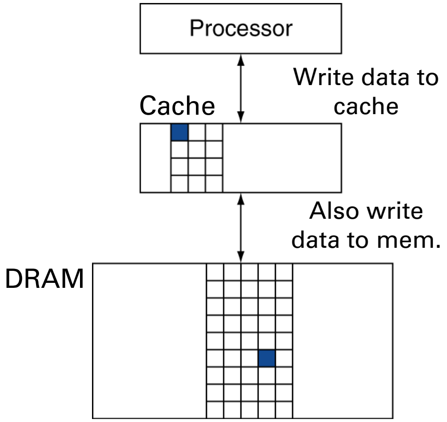
Handling Writes (write-through)

- Suppose on a store instruction,
  - we wrote the data (which is hit on cache) into only the data cache and
  - we do not write the data in the main memory
- After the write into the cache, memory would have a different value from that in the cache

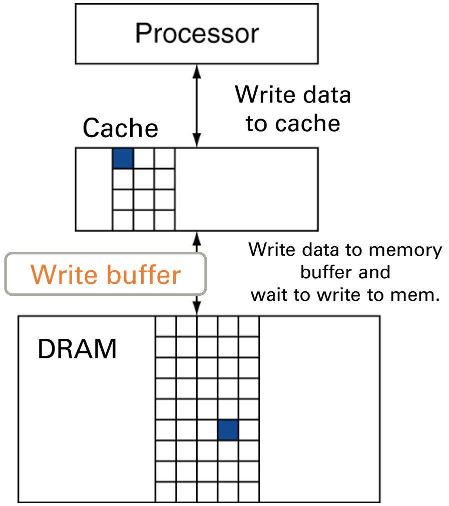
  → the cache and memory are said to be inconsistent
- Two schemes to tackle the inconsistency problem: write-through and write-back

- Write-through scheme
  - A scheme in which writes always update both the cache and
  - the next lower level of the memory hierarchy,
  - ensuring that data are always consistent between the two
- It makes writes take longer
  - E.g., if base CPI = 1, 10% of instructions are stores, and
  - write to memory takes 100 cycles
  - Effective CPI = 1 + 0.1 × 100 = 11
- Write buffer can be used to reduce the memory write latency
  - Temporarily holds data while waiting to be written to memory
  - CPU continues immediately
  - Only stalls on write if write buffer is already full
  - The buffer is of no use when <u>writes are being generated faster</u> than <u>the memory system can accept them</u>

<u>Write-through cache design</u>



<u>Write-through cache w/ memory buffer design</u>



Handling Writes (write-back)

- Write-back scheme
  - Update to cache and write to memory when necessary
  - I.e., a scheme that handles writes by updating values only to the block in the cache, then
  - writing the modified block to the lower level of the hierarchy when the block is replaced

- On data-write hit
  - just update the block in cache
- Keep track of whether each block is dirty
  - Need a dirty bit for each block (next to the Valid Bit)
  - When a dirty block (modified block) is replaced
  - Write it back to memory

- Can use a write buffer
  - to allow replacing block to be read first (halves the miss penalty)
  - The modified block is moved to a write-back buffer associated with the cache *while* the requested block is read from memory

Two Write Miss Policies

- Writes introduce several complications into caches that are not present for reads

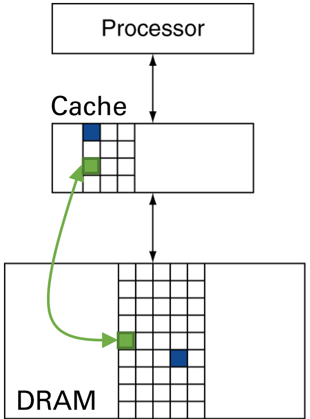
1. Write allocate

- ➢ Allocate a block in cache
- ➢ The block is fetched from memory (similar to a read miss) and
- ➢ then the appropriate portion of the block is overwritten on cache and memory
- ➢ Simple (easy to implement)

2. No write allocate

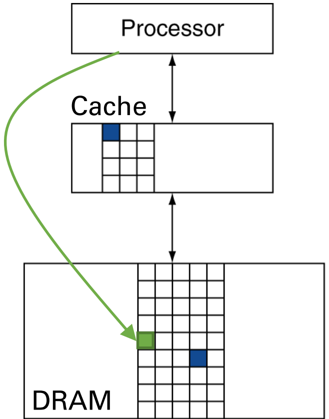
- ➢ To update the portion of the block in memory
- ➢ but not put it in the cache
- ➢ It is useful since programs often write a whole block before reading it (e.g., initialization) without fetching data first (unnecessary)
- ➢ Reduce miss rate

Write allocate policy



- Write-through and write-back caches can use either one of the two policies
- It is a common practice that
  - ✓ write-through cache adopts no write allocate policy
  - ✓ write-back cache uses write allocate policy

No write allocate policy



Example: Caches of MIPS Intrinsity FastMATH

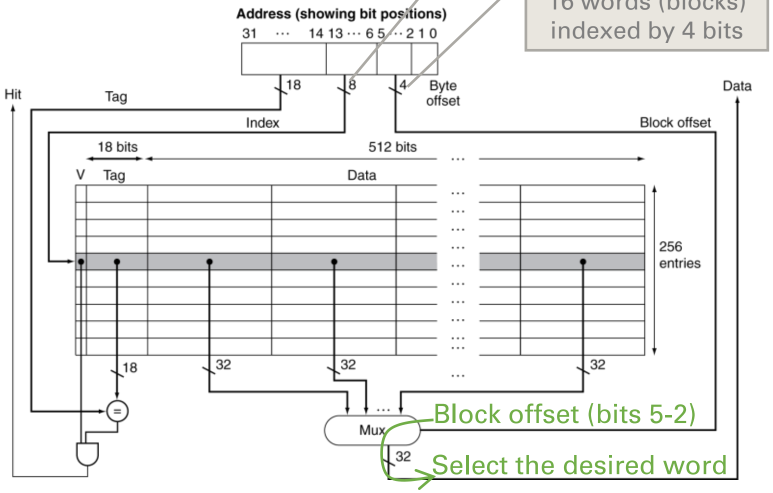
- An embedded MIPS processor
  - ➢12-stage pipeline
  - ➢Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - ➢Each 16KB: 256 blocks × 16-word blocks
  - ➢D-cache: write-through or write-back (OS decides)
  - ➢A one-entry write buffer
- SPEC2000 miss rates
  - ➢I-cache: 0.4%
  - ➢D-cache: 11.4%
  - ➢Weighted average: 3.2%

Steps for a read request to either cache

- Send the address to the appropriate cache. The address comes either from the PC (for an instruction) or from the ALU (for data)
- If the cache signals hit, the requested word is available on the data lines. Since there are 16 words in the desired block, we need to select the right one. A block index field is used to control the multiplexor (shown at the bottom of the figure), which selects the requested word from the 16 words in the indexed block write-through cache adopts no write allocate policy
- If the cache signals miss, we send the address to the main memory. When the memory returns with the data, we write it into the cache and then read it to fulfill the request

256 blocks indexed by 8 bits

16 words (blocks) indexed by 4 bits



Block offset (bits 5-2)

Select the desired word

FIGURE 5.12 The 16 KiB caches in the Intrinsity FastMATH each contain 256 blocks with 16 words per block. Note that the address size for this computer is just 32 bits. The tag field is 18 bits wide and the index field is 8 bits wide, while a 4-bit field (bits 5–2) is used to index the block and select the word from the block using a 16-to-1 multiplexor. In practice, to eliminate the multiplexor, caches use a separate large RAM for the data and a smaller RAM for the tags, with the block offset supplying the extra address bits for the large data RAM. In this case, the large RAM is 32 bits wide and must have 16 times as many words as blocks in the cache

Split cache: A scheme in which a level of the memory hierarchy is composed of two independent caches that operate in parallel with each other, with one handling instructions and one handling data

Cache Performance

- CPU time can be spent on:

1. executing the program
   - → Program execution cycles
   - ➢ Includes cache hit time
2. waiting for memory system
   - → Memory stall cycles
   - ➢ Mainly from cache misses
   - ➢ With simplifying assumptions that stall cycles come from reads and writes

- This section explores two different techniques for improving cache performance

1) One focuses on reducing the miss rate
   - ➢ by reducing the probability that two distinct memory blocks will contend for the same cache location
2) The second technique reduces the miss penalty
   - ➢ by adding an additional level to the hierarchy (first appeared in '90s)
   - ➢ Called *multilevel caching*

$$\text{CPU time} = (\text{CPU execution cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\begin{aligned}\text{Memory stall cycles} \\ &= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty} \\ &= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\end{aligned}$$

- This equation merges the reads/writes into a single equation
- It ignores the *write buffer stall* cycles
  - ✓ Please refer to p. 413~414 in Sec. 5.4 of textbook

Example: Cache Performance Measurement

- Assume the processor
  - has a CPI of 2 (w/ ideal cache)
- The miss rates for
  - I-cache: 2%
  - D-cache: 4%
- The miss penalty is 100 cycles
  - same for read/write misses
- Load & stores are 36% of instructions

- Please compare performance of ideal and actual processor
  - CPI of ideal processor is 2
  - What is the CPI for the above processor setting?

- Assume total instruction count is *I*
- Miss cycles at
  - I-cache: $I \times 0.02 \times 100 = 2I$
  - D-cache: $I \times 0.36 \times 0.04 \times 100 = 1.44I$
- Total number of memory stall cycles
  - $3.44I$ $(= 2I + 1.44I)$
  - More than three cycles of memory stall per instruction
- Actual CPI = 2 + 2 + 1.44 = 5.44

- Actual processor is 2.72 times slower than ideal processor
  - CPI for Ideal CPU is 2
  - CPI for Actual CPU is 5.44
  - 5.44/2 =2.72

- For detailed descriptions…
  - ✓ Please refer to p. 414~415 in Sec. 5.4 of textbook

Average Memory Access Time

- Average memory access time (AMAT)

→ AMAT = Hit time + Miss rate × Miss penalty

- Assess the *time* to access data for both *hits* and *misses*
  - ➢A way to examine alternative cache designs
  - ➢Average time to access memory considering both hits and misses and the frequency of different accesses

Example

- CPU with 1ns clock cycle time, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%

- AMAT = 1 + 0.05 × 20 = 2ns (2 clock cycles)

- For detailed descriptions…
  - ✓ Please refer to p. 415~416 in Sec. 5.4 of textbook

Cache Design
2: Fully Associative

- One extreme of placement scheme: direct mapped
  - There is a direct mapping from any block address in memory to a single location in the upper level of the hierarchy
- The other extreme scheme: fully associative
  - A cache structure in which a block can be placed in *any* location in the cache
  - I.e., a block in memory may be associated with any entry in the cache
  - Thus, it requires all entries to be searched at once
- The search is done in parallel with a comparator associated with each cache entry
  - These comparators significantly increase the hardware cost,
  - effectively making fully associative placement practical
  - only for caches with small numbers of blocks

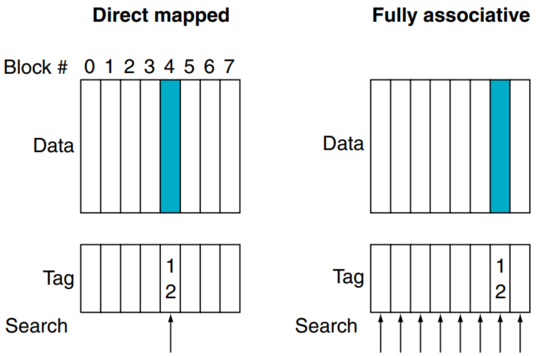


FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement. In directmapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks

Cache Design
3: Set Associative

- The middle range of designs between the above two extremes → set associative
  - ➢A cache that has a fixed number of locations (at least two) where each block can be placed
- *N*-way set associative
  - ➢A block in memory can put in a set of cache entries
  - ➢Each set contains *N* entries
- Calculate which set for a block
  - → #Block number modulo #Sets in cache
- Search all entries (tags) in a given set at once in parallel to find the exact block
  - ➢*N* comparators (less expensive)

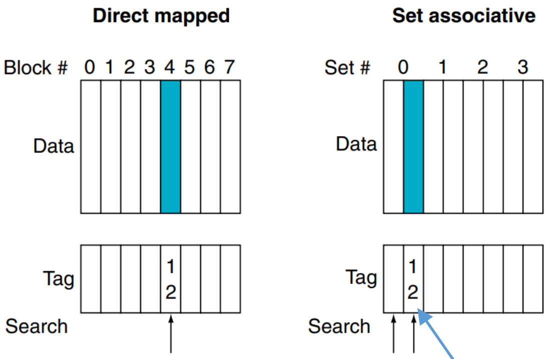


FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement. In directmapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4.
In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks

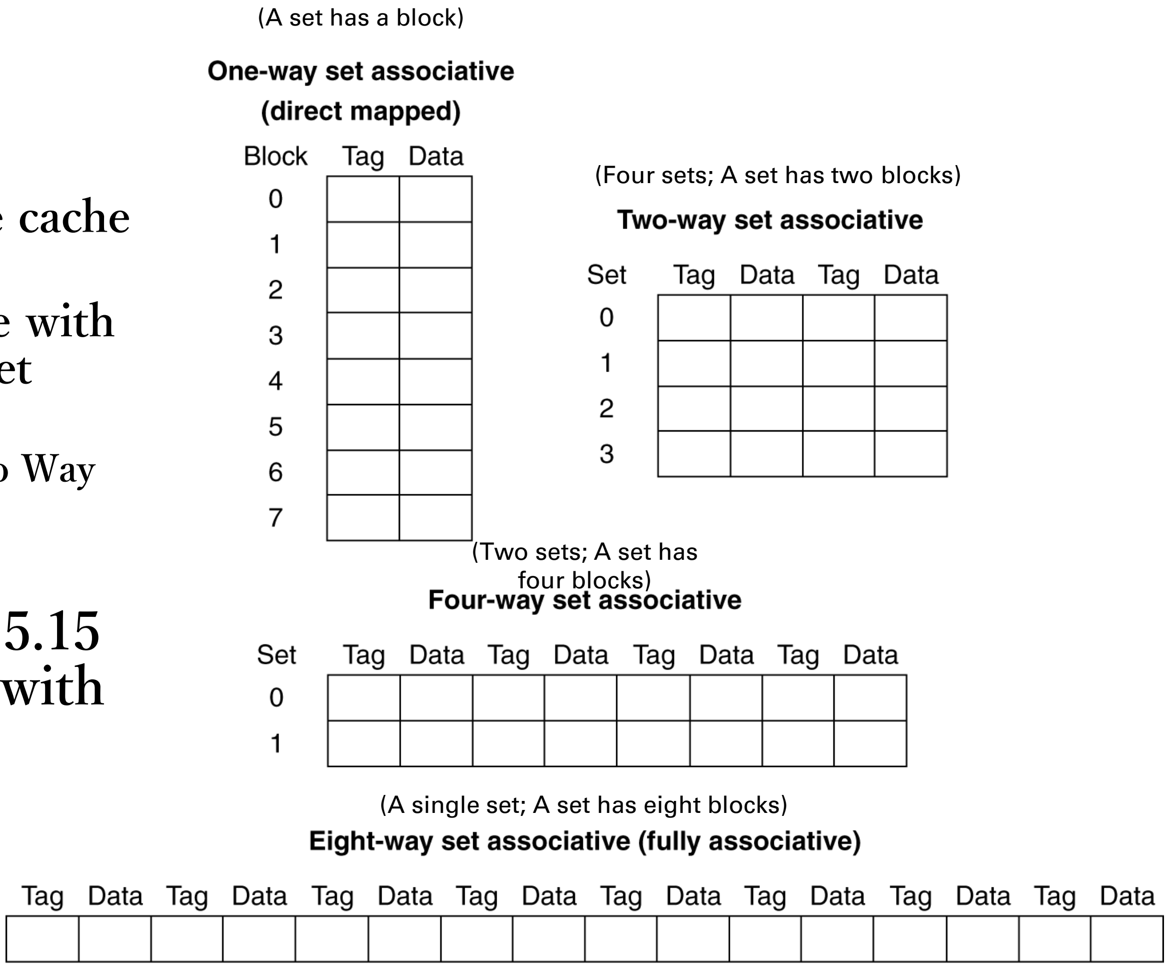
- Given a 2-way cache with a total of 8 entries, please find which cache set for a memory block of address 12
- Total entries = 8, blocks of each set = 2 (two ways)
  - ✓ Cache sets = 4 (sets)
  - ✓ Mem. Block address % #Cache set = 12 % 4 = 0
  - ✓ The block of address 12 is assigned to Set 0

"Number of blocks in the cache" for directed map design

Example:
M-way Set Associative Caches

- Set associative cache variants
  - One-way set associative cache (direct-mapped cache)
  - A fully associative cache with m entries (is a *m*-way set associative cache)
    - Entry number is equal to Way number

- The examples in Fig. 5.15 are variants of a cache with eight entries


(A set has a block)

One-way set associative (direct mapped)

(Four sets; A set has two blocks)

Two-way set associative

(Two sets; A set has four blocks)

Four-way set associative

(A single set; A set has eight blocks)

Eight-way set associative (fully associative)

FIGURE 5.15 An eight-block cache configured as direct-mapped, two-way set associative, four-way set associative, and fully associative. The total size of the cache in blocks is equal to the number of sets times the associativity. Thus, for a fixed cache size, increasing the associativity decreases the number of sets while increasing the number of elements per set. With eight blocks, an eight-way set associative cache is the same as a fully associative cache

Example: Misses for Variant Cache Designs (1/2)

- Assume the address sequence of memory blocks:
  - 0, 8, 0, 6, 8

- There are three four-block caches
  - Each block is *one* word
  1. Direct mapped cache
  2. Two-way set associative cache (w/ LRU replacement policy)
  3. Fully associative cache

- Please find the number of misses for the three caches

Example:
Misses for Variant Cache Designs (2/2)

- Direct mapped cache
  - Four blocks; 5 misses

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Block addr. % 4
(e.g., 6 % 4 = 2)

- Two-way associative cache
  - Four blocks (two sets)
  - 4 misses & 1 hit
  - Using LRU policy to select the victim block when a set is full

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

Set 0 is full @ the 3rd access

Block addr. % 2
(e.g., 6 % 2 = 0)

- The least recently used (LRU) block is Mem[8], whereas Mem[0] is the most recently used block
- Mem[8] is replaced with the incoming block Mem[6]

- Fully associative cache
  - Four blocks in a single set
  - 3 misses & 2 hits

Block address		Hit/miss	Cache content after access			
			Block 0	Block 1	Block 2	Block 3
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

Any mem. block can be stored in any cache block

- Access patterns, cache size, and associativity affects the delivered performance
- What are the misses when the access sequence is 21, 22, 23, 24, 25?

Summary of the Three Caches

- Direct mapped cache
  - Could has higher conflict misses for certain memory access patterns
  - Conflict miss: Different memory blocks contend for the same cache block
- Fully associative cache
  - Can minimize the conflict misses
  - At the cost of the high HW cost for comparators
- Set associative cache
  - A middle way to balance the hardware cost and performance
- The choice among the three schemes
  - will depend on the cost of a miss vs.
  - the cost of implementing associativity,
  - both in time and in extra hardware

Example:
A Four-Way Set-Associative Cache

- Cache size: 1K blocks
- Block size: 4 bytes
  - ➢One word
- Set numbers: 256
  - ➢1,024 blocks/4 ways
- Index bits for addressing the sets: 8
  - ➢256 = 2^8
- Byte offset: 2
- Tag bits: 22
  - ➢32-8-2

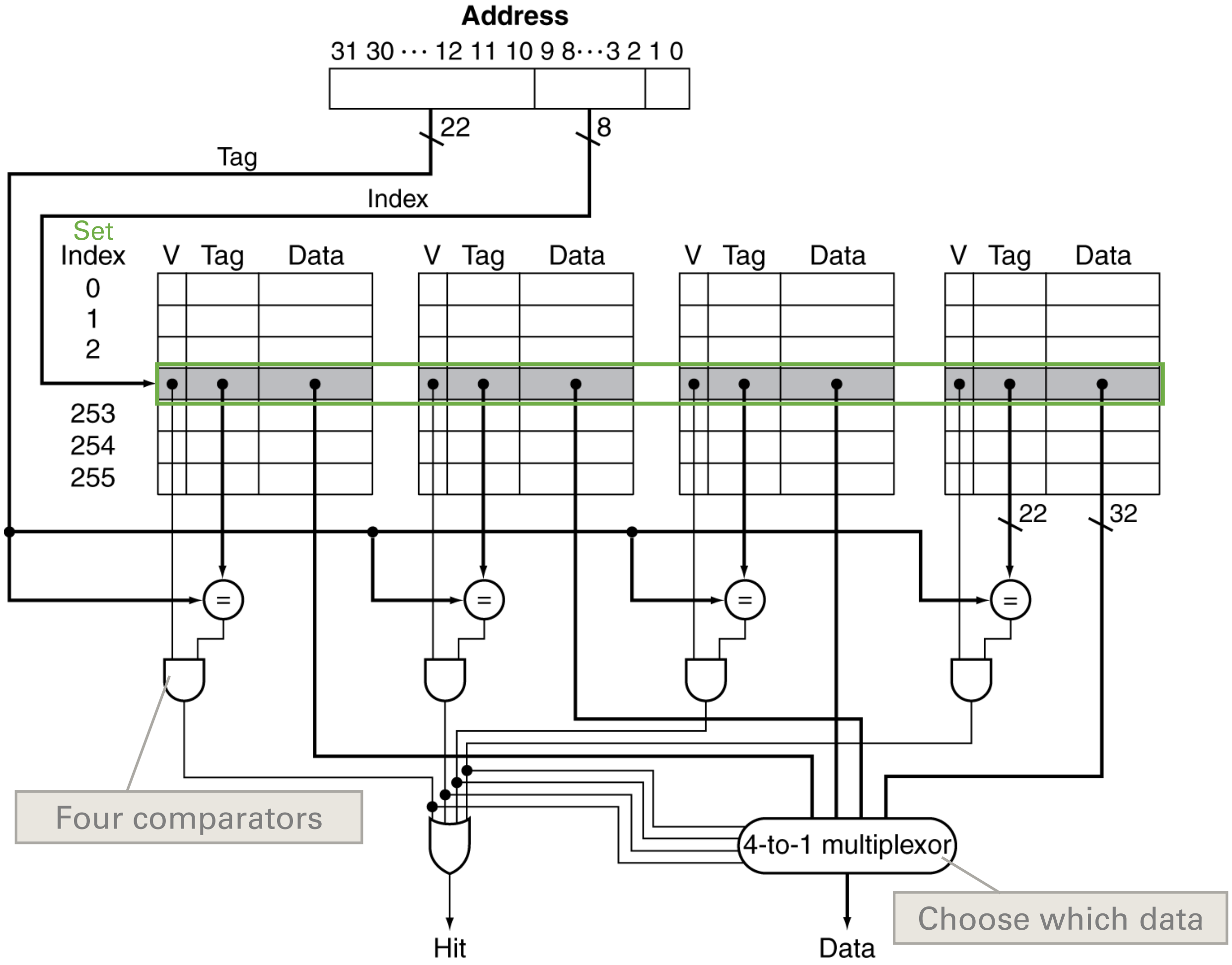


FIGURE 5.18 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor. The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor

- More about the bit length of tag and index can refer to p. 421 and p. 423 (example) of the textbook

Performance Summary

- When CPU performance increased
  - ➢Miss penalty becomes more significant
- Decreasing base CPI
  - ➢Greater proportion of time spent on memory stalls
- Increasing clock rate
  - ➢Memory stalls account for more CPU cycles
- Cannot neglect cache behavior when evaluating system performance

Cache Block Replacement

- When a miss occurs…
- in a direct-mapped cache,
  - ➢ the requested block can go in exactly one position, and the block occupying that position must be replaced
- in an associative cache,
  - ➢ we have a choice of where to place, and hence a choice of which block to replace
- in a fully associative cache,
  - ➢ all blocks are candidates for replacement

- Furthermore, in a set-associative cache,
  - ➢ we must choose among the blocks in the selected set

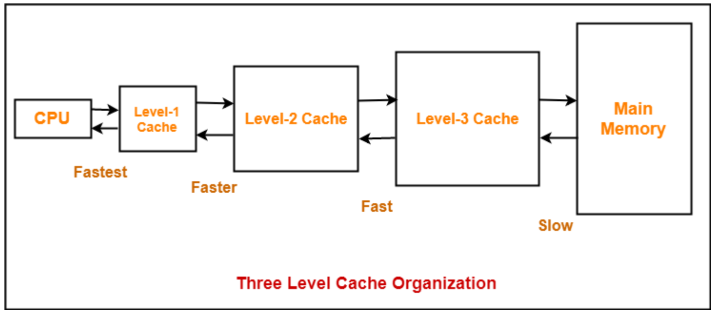
- Least recently used (LRU) policy is often used to choose the block to be replaced
  - ➢ A replacement scheme in which the block replaced is the one that has been unused for the longest time
  - ➢ Done by keeping tracks of when each element in a set was used relative to the other elements in the set
  - ➢ It is easy to be done for 2-way set-associative case (using a accessing bit); it gets harder when associativity increases

Multilevel Caches

- Additional levels of caching are used
  - to close the gap further between the fast clock rates of modern processors and the increasingly long time required to access DRAMs

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
  - Reduce writes on memory (miss penalty)
- Main memory services L-2 cache misses

- Some high-end systems include L-3 cache as shown in the right figure



Three Level Cache Organization

Courtesy of ImageSource

Example: Multilevel Cache Performance Evaluation

- Suppose a processor with a primary cache
  - ➢Base CPI = 1, clock rate = 4GHz (0.25ns per clock cycle)
  - ➢Main memory access time = 100ns (include all miss handling)
  - ➢Miss rate per instruction = 2%
- The CPI for the processor with a primary cache
  - ➢Miss penalty in cycles = 100ns/0.25ns = 400 cycles
  - ➢Effective CPI
    = 1 (base) + 0.02 × 400 (memory-stall cycle per instruction)
    = 9
- How much faster will the processor be if a level-2 cache is added?
  - ➢The L2 cache has a 5ns access time for a hit/mis
  - ➢is large enough to reduce the miss rate to memory to 0.5%
- The CPI for the processor with the two levels of caches
  - ➢L1 miss and L2 hit
    Penalty = 5ns/0.25ns = 20 cycles
  - ➢L1 miss and L2 miss
    100ns/0.25ns = 400 cycles
  - ➢Effective CPI
    = 1 +0.02 × 20 (L1 miss * L2 hit) + 0.005 × 400 (L1 and L2 misses)
    = 3.4
- Performance difference between the two designs
  - ➢w/ only L1 cache and w/ two levels of caches
  - ➢Performance ratio = 2.6 (=9/3.4)

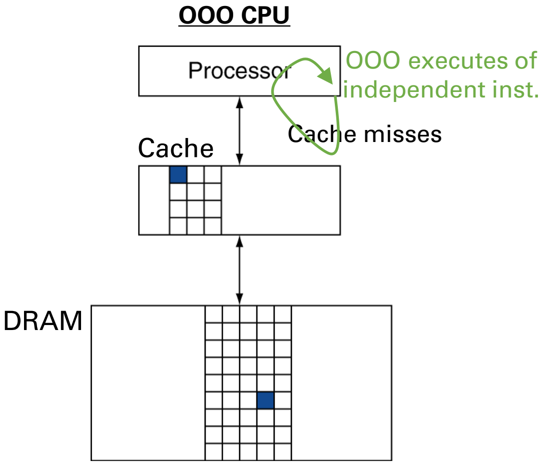
Design Considerations of Multilevel Caches

- Level-1 cache
  - Focus on minimal hit time
- Level-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - Level-1 cache usually smaller than a single cache
  - Level-1 block size smaller than level-2 block size

Interactions with Advanced CPUs

- Out-of-order CPUs can execute instructions during cache misses
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
  - Independent instructions continue

- No general way to calculate overlapped miss latency
  - It requires simulation of the processor and the memory hierarchy to evaluate of memory hierarchies performance



- More about the idea of overlapped misses can refer to p. 429 and p. 430 (elaborations) of the textbook

Hardware Software Interactions

- To understand the performance of programs on modern computers
  - it is critical to understand the behavior (and organization) of the memory hierarchy
- Many software optimizations were invented to dramatically improve performance
  - by reusing data within the cache and
  - hence lower miss rates due to improved temporal locality

- Taking array accessing as an example,
  - we can get good performance from the memory system
  - if we store the array in memory so that accesses to the array are sequential in memory
- If fact, as for multiple arrays, however,
  - some arrays accessed by rows and some by columns

Blocked Data Accesses (Data Tiles)

- *Blocked* algorithms operate on submatrices or blocks
  - instead of operating on entire rows or columns of an array
  - They are commonly used to maximize accesses to cached data (improving locality) before the data are replaced
- The concept of the access patterns of arrays A, B, and C in Fig. 5.20
  - A dark shade indicates a recent access,
  - a light shade indicates an older access, &
  - white means not yet accessed

- The number of capacity misses clearly depends on N and the cache size
- If it can hold all three N-by-N matrices, then all is well, provided there are no cache conflicts

The inner loops of DGEMM
(Lines 4 through 9 of Figure 3.22 in Chapter 3; p. 427)

```
for (int j = 0; j < n; ++j)  {
  double cij = C[i+j*n];                    /* cij = C[i][j] */
  for( int k = 0; k < n; k++ )
    cij += A[i+k*n] * B[k+j*n];             /* cij += A[i][k]*B[k][j] */
  C[i+j*n] = cij;                           /* C[i][j] = cij */
}
```

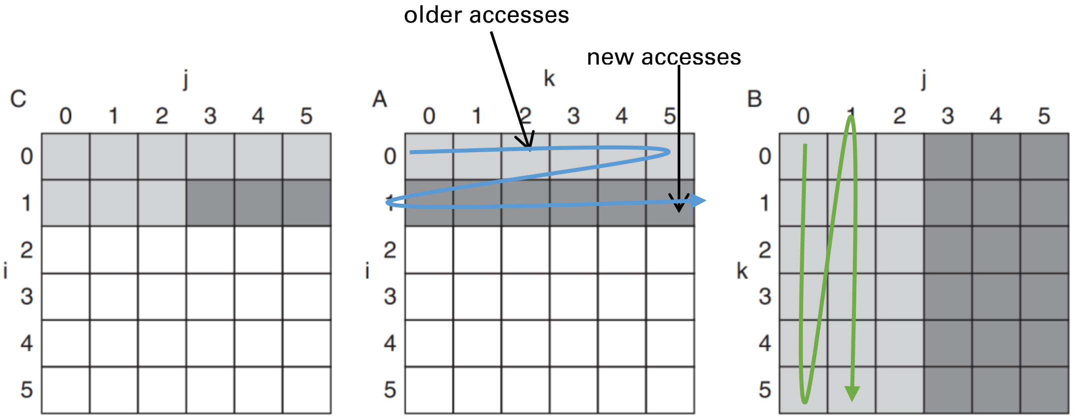
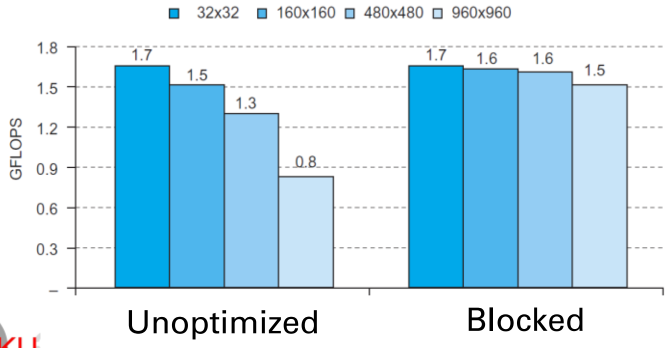


FIGURE 5.20 A snapshot of the three arrays C, A, and B when N = 6 and i = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to Figure 5.22, elements of A and B are read repeatedly to calculate new elements of C. The variables i, j, and k are shown along the rows or columns used to access the arrays

Blocked Data Accesses (Cont’d)

- Fig. 5.22 illustrates the accesses to the three arrays using blocking
  - ➢Same access patterns, but within smaller regions
  - ➢Refer to p. 427~429 in textbook for details
- Looking only at capacity misses,
  - ➢the total number of memory words accessed is $2\ N^3\ /\text{BLOCKSIZE} + N^2$
  - ➢Originally, $2\ N^3 + N^2$ memory words are accessed for N^3 operations
- This total is an improvement by about a factor of BLOCKSIZE
  - ➢Depending on the computer and size of the matrices, blocking can improve performance by 2x (to 10x)
  - ➢E.g., a cache block size can accommodate BLOCKSIZE data

- Blocked version has better computing efficiency from small to large data



- Blocked code enjoys both spatial and temporal locality

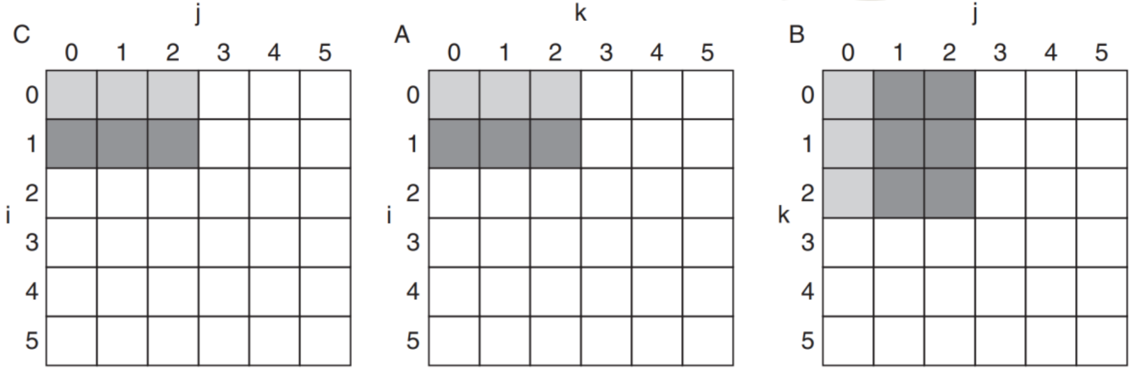


FIGURE 5.22 The age of accesses to the arrays C, A, and B when BLOCKSIZE = 3. Note that, in contrast to Figure 5.20, fewer elements are accessed

§5.8 A Common Framework for Memory Hierarchies

Memory Hierarchy

- Common principles apply at all levels of the memory hierarchy

- The key design decisions at each level in the hierarchy

  (Based on notions of caching)

  - ➢Block placement
  - ➢Finding a block
  - ➢Replacement on a miss
  - ➢Write policy

- Question 1: Where can a block be placed?
  Answer: One place (direct mapped), a few places (set associative), or any place (fully associative)
- Question 2: How is a block found?
  Answer: There are four methods: indexing (as in a direct-mapped cache), limited search (as in a set-associative cache), full search (as in a fully associative cache), and a separate lookup table (as in a page table)
- Question 3: What block is replaced on a miss?
  Answer: Typically, either the least recently used or a random block
- Question 4: How are writes handled?
  Answer: Each level in the hierarchy can use either write-through or write-back.

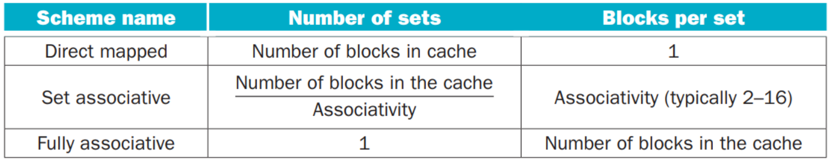
Block Placement

- Different schemes can be thought of as variations on a set-associative scheme
  - where the number of sets and the number of blocks per set varies

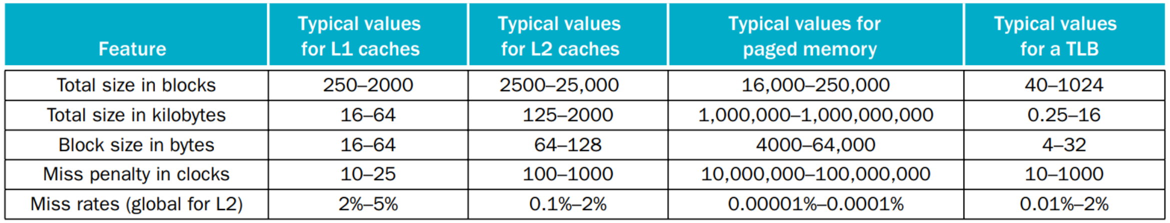
Determined by associativity

- Higher associativity reduces miss rate
  - ✓ But, increases complexity, cost, and access time

- Direct mapped (1-way associative)
  - One choice for placement
- N-way set associative
  - N choices within a set
- Fully associative
  - Any location



Scheme name	Number of sets	Blocks per set
Direct mapped	Number of blocks in cache	1
Set associative	$\frac{\text{Number of blocks in the cache}}{\text{Associativity}}$	Associativity (typically 2–16)
Fully associative	1	Number of blocks in the cache



Feature	Typical values for L1 caches	Typical values for L2 caches	Typical values for paged memory	Typical values for a TLB
Total size in blocks	250–2000	2500–25,000	16,000–250,000	40–1024
Total size in kilobytes	16–64	125–2000	1,000,000–1,000,000,000	0.25–16
Block size in bytes	16–64	64–128	4000–64,000	4–32
Miss penalty in clocks	10–25	100–1000	10,000,000–100,000,000	10–1000
Miss rates (global for L2)	2%–5%	0.1%–2%	0.00001%–0.0001%	0.01%–2%

FIGURE 5.33 The key quantitative design parameters that characterize the major elements of memory hierarchy in a computer. These are typical values for these levels as of 2020. Although the range of values is wide, this is partially because many of the values that have shifted over time are related; for example, as caches become larger to overcome larger miss penalties, block sizes also grow. While not shown, server microprocessors today also have L3 caches, which can be 4 to 50 MiB and contain many more blocks than L2 caches. L3 caches lower the L2 miss penalty to 30 to 40 clock cycles

Find a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
N-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

- Hardware caches
  - ➢ Reduce comparisons to reduce cost
  - ➢ The choice among the above designs will depend on
    - ❖ the cost of a miss versus the cost of implementing associativity, both in time and in extra hardware
  - ➢ Set-associative placement is often used for caches and TLBs, where the access combines indexing and the search of a small set
- Virtual memory
  - ➢ Full associativity is beneficial, since misses are very expensive
  - ➢ Full associativity allows software to use sophisticated replacement schemes that are designed to reduce the miss rate
  - ➢ The full map can be easily indexed with no extra hardware and no searching required

Replacement

- Two major methods of selecting an entry to replace on a miss

- Least recently used (LRU)
  - The block replaced is the one that has been unused for the longest time
  - Complex and costly hardware for high associativity (> 4 sets)
    - Approximated via, for example, 1 reference bit
- Random
  - Candidate blocks are randomly selected, possibly using some hardware assistance
  - Performance close to LRU (could be better than approximated LRU)
  - Easy to implement
  - E.g., for a two-way set-associative cache, random replacement has a miss rate about 1.1 times higher than LRU replacement
  - The performance gap between random and LRU becomes smaller when the cache size grows

- For virtual memory
  - LRU approximation with hardware support

Write Policy

- Write-through
  - Update both upper and lower levels
  - The information is written to both the block in the cache and the block in the lower level of the memory hierarchy
  - Easier to implement & simplifies replacement, but may require write buffer
- Write-back
  - Update upper level only, and update lower level when block is replaced
  - The information is written just to the block in the cache
  - The modified block is written to the lower level of the hierarchy only when it is replaced
  - Multiple writes within a block require only one write to the lower level in the hierarchy
  - Need to keep more state

- The above two schemes are possible
  - between cache and main memory
- For virtual memory
  - Only write-back is feasible
  - because of the long latency of a write to the lower level of the hierarchy

The Three Cs (Cache Misses)

- Compulsory misses
  - These are cache misses caused by *the first access to a block that has never been in the cache*. These are also called cold-start misses
- Capacity misses
  - These are cache misses caused when the cache cannot contain all the blocks needed during execution of a program. Capacity misses occur when *blocks are replaced and then later retrieved*
- Conflict misses
  - These are cache misses that occur in set-associative or direct-mapped caches when *multiple blocks compete for the same set*
  - Conflict misses are those misses in a direct-mapped or set-associative cache that are eliminated in a fully associative cache of the same size
  - These cache misses are also called collision misses
- There is another type of cache misses: coherence miss
  - Happens in a multicore processor

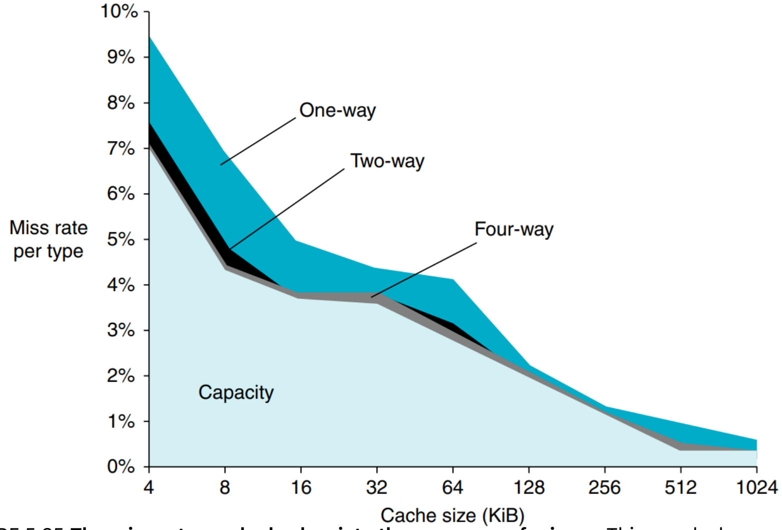


FIGURE 5.35 **The miss rate can be broken into three sources of misses**. This graph shows the total miss rate and its components for a range of cache sizes. These data are for the SPEC CPU2000 integer and floating-point benchmarks and are from the same source as the data in Figure 5.34. The **compulsory miss** component is 0.006% and cannot be seen in this graph. The next component is the **capacity miss** rate, which depends on cache size. The **conflict** portion, which depends both on associativity and on cache size, is shown for a range of associativities from one-way to eight-way. In each case, the labeled section corresponds to the increase in the miss rate that occurs when the associativity is changed from the next higher degree to the labeled degree of associativity. For example, the section labeled two-way indicates the additional misses arising when the cache has associativity of two rather than four. Thus, the difference in the miss rate incurred by a direct-mapped cache versus a fully associative cache of the same size is given by the sum of the sections marked four-way, two-way, and one-way. The difference between eight-way and four-way is so small that it is difficult to see on this graph

Cache Design Challenges

- The challenge in designing memory hierarchies is that
  - every change that potentially improves the miss rate can also negatively affect overall performance
  - as Fig. 5.36 summarizes
- This combination of positive and negative effects is
  - what makes the design of a memory hierarchy interesting

Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

Figure 5.36 Memory hierarchy design challenges

The Rest of this Chapter

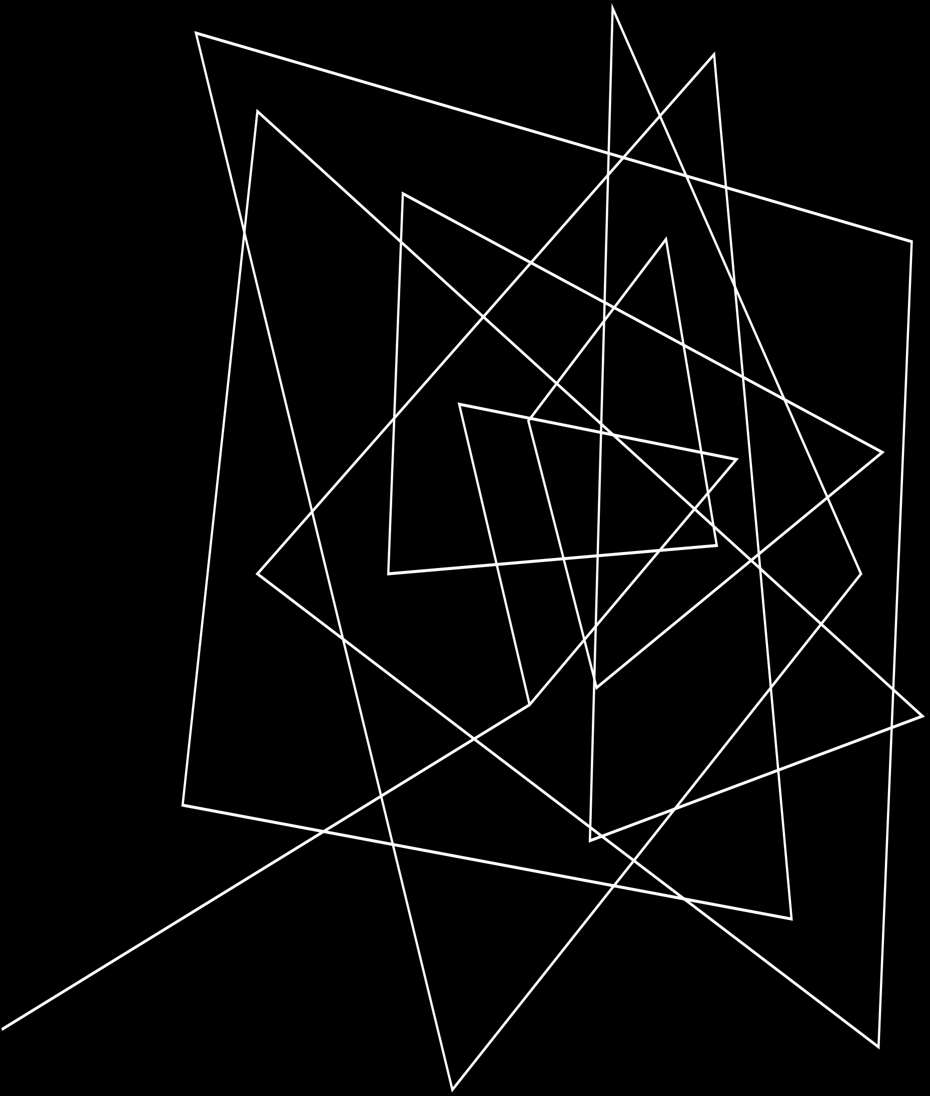
- Because of time limitation, the contents are not covered...
- Sec. 5.9 Using a Finite-State Machine to Control a Simple Cache
  - Focuses on the design of a cache controller with finite-state machines
- Sec. 5.10 Parallelism and Memory Hierarchy: Cache Coherence
  - Talks about the challenging issue of caching shared data of a common physical address space for a multiprocessor on a single chip (cache coherence, snooping protocols)
- Sec. 5.16 Fallacies and Pitfalls
  - It will be useful if you have some time to read this section

- Sec. 5.10 is important as the multicore processors are prevalent nowadays
  - You can read it if you are interested in this topic
  - This advanced topic is often covered in a graduate course for advanced computer architecture

Concluding Remarks

§5.17 Concluding Remarks

- Fast memories are small, large memories are slow
  - ➢We really want fast, large memories ☹
  - ➢Caching gives this illusion ☺
- Principle of locality
  - ➢Programs use a small part of their memory space frequently
- Memory hierarchy
  - ➢L1 cache ↔ L2 cache ↔ ⋯ ↔ DRAM memory ↔ disk
- Memory system design is critical for multiprocessors



Questions?