CHAPTER 4

CACHE MEMORY

Learning Objectives

After studying this chapter, you should be able to:

- Present an overview of the main characteristics of computer memory systems and the use of a memory hierarchy.
- Describe the basic concepts and intent of cache memory.
- Discuss the key elements of cache design.
- Distinguish among direct mapping, associative mapping, and set-associative mapping.
- Explain the reasons for using multiple levels of cache.
- Understand the performance implications of multiple levels of memory.

Although seemingly simple in concept, computer memory exhibits perhaps the widest range of type, technology, organization, performance, and cost of any feature of a computer system. No single technology is optimal in satisfying the memory requirements for a computer system. As a consequence, the typical computer system is equipped with a hierarchy of memory subsystems, some internal to the system (directly accessible by the processor) and some external (accessible by the processor via an I/O module).

This chapter and the next focus on internal memory elements, while Chapter 6 is devoted to external memory. To begin, the first section examines key characteristics of computer memories. The remainder of the chapter examines an essential element of all modern computer systems: cache memory.

4.1 COMPUTER MEMORY SYSTEM OVERVIEW

Characteristics of Memory Systems

The complex subject of computer memory is made more manageable if we classify memory systems according to their key characteristics. The most important of these are listed in Table 4.1.

The term **location** in Table 4.1 refers to whether memory is internal or external to the computer. Internal memory is often equated with main memory, but there are other forms of internal memory. The processor requires its own local memory, in the form of registers (e.g., see Figure 2.3). Further, as we will see, the control unit portion of the processor may also require its own internal memory. We will defer discussion of these latter two types of internal memory to later chapters. Cache is another form of internal memory. External memory consists of peripheral storage devices, such as disk and tape, that are accessible to the processor via I/O controllers.

An obvious characteristic of memory is its **capacity**. For internal memory, this is typically expressed in terms of bytes (1 byte = 8 bits) or words. Common word lengths are 8, 16, and 32 bits. External memory capacity is typically expressed in terms of bytes.

Table 4.1 Key Characteristics of Computer Memory Systems

Location	**Performance**
Internal (e.g., processor registers, cache, main memory)	Access time
External (e.g., optical disks, magnetic disks, tapes)	Cycle time
Capacity	Transfer rate
Number of words	**Physical Type**
Number of bytes	Semiconductor
Unit of Transfer	Magnetic
Word	Optical
Block	Magneto-optical
Access Method	**Physical Characteristics**
Sequential	Volatile/nonvolatile
Direct	Erasable/nonerasable
Random	**Organization**
Associative	Memory modules

A related concept is the **unit of transfer**. For internal memory, the unit of transfer is equal to the number of electrical lines into and out of the memory module. This may be equal to the word length, but is often larger, such as 64, 128, or 256 bits. To clarify this point, consider three related concepts for internal memory:

- **Word:** The "natural" unit of organization of memory. The size of a word is typically equal to the number of bits used to represent an integer and to the instruction length. Unfortunately, there are many exceptions. For example, the CRAY C90 (an older model CRAY supercomputer) has a 64-bit word length but uses a 46-bit integer representation. The Intel x86 architecture has a wide variety of instruction lengths, expressed as multiples of bytes, and a word size of 32 bits.
- **Addressable units:** In some systems, the addressable unit is the word. However, many systems allow addressing at the byte level. In any case, the relationship between the length in bits A of an address and the number N of addressable units is $2^A = N$.
- **Unit of transfer:** For main memory, this is the number of bits read out of or written into memory at a time. The unit of transfer need not equal a word or an addressable unit. For external memory, data are often transferred in much larger units than a word, and these are referred to as blocks.

Another distinction among memory types is the **method of accessing** units of data. These include the following:

- **Sequential access:** Memory is organized into units of data, called records. Access must be made in a specific linear sequence. Stored addressing information is used to separate records and assist in the retrieval process. A shared read–write mechanism is used, and this must be moved from its current location to the desired location, passing and rejecting each intermediate record. Thus, the time to access an arbitrary record is highly variable. Tape units, discussed in Chapter 6, are sequential access.
- **Direct access:** As with sequential access, direct access involves a shared read–write mechanism. However, individual blocks or records have a unique

address based on physical location. Access is accomplished by direct access to reach a general vicinity plus sequential searching, counting, or waiting to reach the final location. Again, access time is variable. Disk units, discussed in Chapter 6, are direct access.

- **Random access:** Each addressable location in memory has a unique, physically wired-in addressing mechanism. The time to access a given location is independent of the sequence of prior accesses and is constant. Thus, any location can be selected at random and directly addressed and accessed. Main memory and some cache systems are random access.
- **Associative:** This is a random access type of memory that enables one to make a comparison of desired bit locations within a word for a specified match, and to do this for all words simultaneously. Thus, a word is retrieved based on a portion of its contents rather than its address. As with ordinary random-access memory, each location has its own addressing mechanism, and retrieval time is constant independent of location or prior access patterns. Cache memories may employ associative access.

From a user's point of view, the two most important characteristics of memory are capacity and **performance**. Three performance parameters are used:

- **Access time (latency):** For random-access memory, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-random-access memory, access time is the time it takes to position the read–write mechanism at the desired location.
- **Memory cycle time:** This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively. Note that memory cycle time is concerned with the system bus, not the processor.
- **Transfer rate:** This is the rate at which data can be transferred into or out of a memory unit. For random-access memory, it is equal to 1/(cycle time). For non-random-access memory, the following relationship holds:

$$T_n = T_A + \frac{n}{R} \tag{4.1}$$

where

T_n = Average time to read or write n bits

T_A = Average access time

n = Number of bits

R = Transfer rate, in bits per second (bps)

A variety of **physical types** of memory have been employed. The most common today are semiconductor memory, magnetic surface memory, used for disk and tape, and optical and magneto-optical.

Several **physical characteristics** of data storage are important. In a volatile memory, information decays naturally or is lost when electrical power is switched off. In a nonvolatile memory, information once recorded remains without deterioration until deliberately changed; no electrical power is needed to retain information. Magnetic-surface memories are nonvolatile. Semiconductor memory (memory on integrated circuits) may be either volatile or nonvolatile. Nonerasable memory cannot be altered, except by destroying the storage unit. Semiconductor memory of this type is known as *read-only memory* (ROM). Of necessity, a practical nonerasable memory must also be nonvolatile.

For random-access memory, the **organization** is a key design issue. In this context, *organization* refers to the physical arrangement of bits to form words. The obvious arrangement is not always used, as is explained in Chapter 5.

The Memory Hierarchy

The design constraints on a computer's memory can be summed up by three questions: How much? How fast? How expensive?

The question of how much is somewhat open ended. If the capacity is there, applications will likely be developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reasonable in relationship to other components.

As might be expected, there is a trade-off among the three key characteristics of memory: capacity, access time, and cost. A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

- Faster access time, greater cost per bit;
- Greater capacity, smaller cost per bit;
- Greater capacity, slower access time.

The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with short access times.

The way out of this dilemma is not to rely on a single memory component or technology, but to employ a **memory hierarchy**. A typical hierarchy is illustrated in Figure 4.1. As one goes down the hierarchy, the following occur:

a. Decreasing cost per bit;

b. Increasing capacity;

c. Increasing access time;

d. Decreasing frequency of access of the memory by the processor.

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization

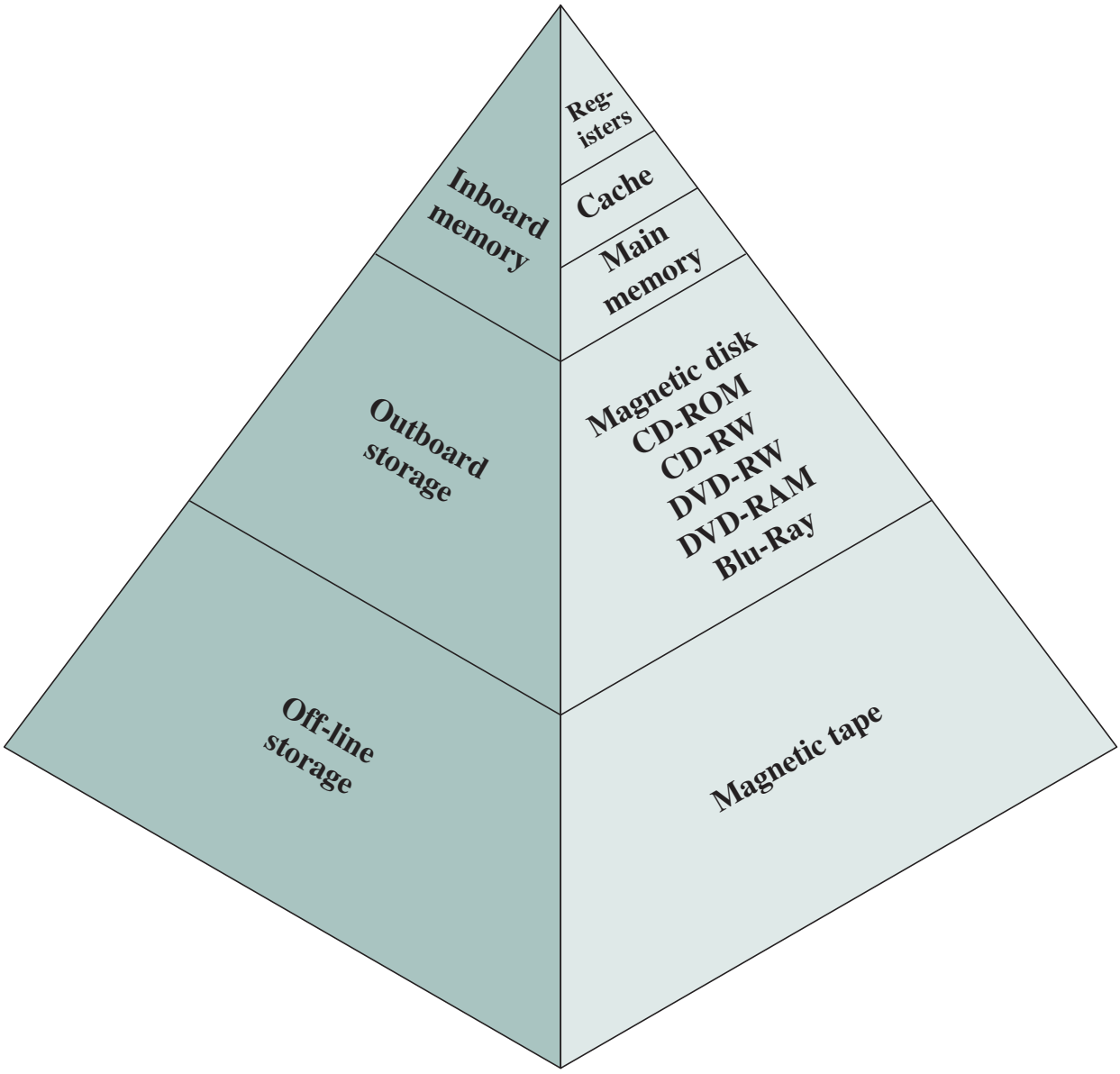


Figure 4.1 The Memory Hierarchy

is item (d): decreasing frequency of access. We examine this concept in greater detail when we discuss the cache, later in this chapter, and virtual memory in Chapter 8. A brief explanation is provided at this point.

The use of two levels of memory to reduce average access time works in principle, but only if conditions (a) through (d) apply. By employing a variety of technologies, a spectrum of memory systems exists that satisfies conditions (a) through (c). Fortunately, condition (d) is also generally valid.

The basis for the validity of condition (d) is a principle known as **locality of reference** [DENN68]. During the course of execution of a program, memory references by the processor, for both instructions and data, tend to cluster. Programs typically contain a number of iterative loops and subroutines. Once a loop or subroutine is entered, there are repeated references to a small set of instructions. Similarly, operations on tables and arrays involve access to a clustered set of data words. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

EXAMPLE 4.1 Suppose that the processor has access to two levels of memory. Level 1 contains 1000 words and has an access time of 0.01 μs; level 2 contains 100,000 words and has an access time of 0.1 μs. Assume that if a word to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the word is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the word is in level 1 or level 2. Figure 4.2 shows the general shape of the curve that covers this situation. The figure shows the average access time to a two-level memory as a function of the hit ratio H, where H is defined as the fraction of all memory accesses that are found in the faster memory (e.g., the cache), T_1 is the access time to level 1, and T_2 is the access time to level 2.[1] As can be seen, for high percentages of level 1 access, the average total access time is much closer to that of level 1 than that of level 2.

In our example, suppose 95% of the memory accesses are found in level 1. Then the average time to access a word can be expressed as

$$(0.95)(0.01\ \mu s) + (0.05)(0.01\ \mu s + 0.1\ \mu s) = 0.0095 + 0.0055 = 0.015\ \mu s$$

The average access time is much closer to 0.01 μs than to 0.1 μs, as desired.

Accordingly, it is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is substantially less than that of the level above. Consider the two-level example already presented. Let level 2

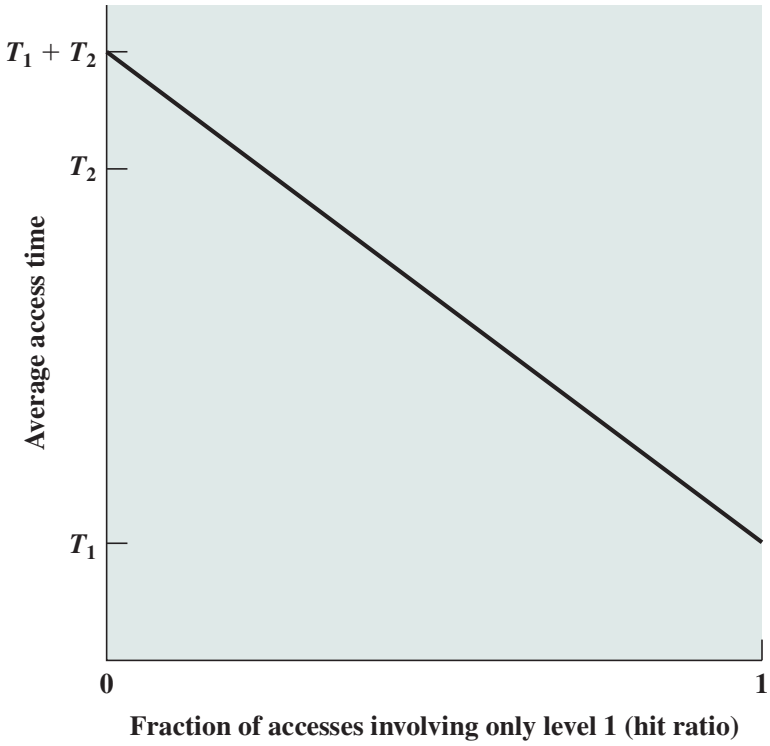


Figure 4.2 Performance of Accesses Involving only Level 1 (hit ratio)

[1] If the accessed word is found in the faster memory, that is defined as a **hit**. A **miss** occurs if the accessed word is not found in the faster memory.

memory contain all program instructions and data. The current clusters can be temporarily placed in level 1. From time to time, one of the clusters in level 1 will have to be swapped back to level 2 to make room for a new cluster coming in to level 1. On average, however, most references will be to instructions and data contained in level 1.

This principle can be applied across more than two levels of memory, as suggested by the hierarchy shown in Figure 4.1. The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor. Typically, a processor will contain a few dozen such registers, although some machines contain hundreds of registers. Main memory is the principal internal memory system of the computer. Each location in main memory has a unique address. Main memory is usually extended with a higher-speed, smaller cache. The cache is not usually visible to the programmer or, indeed, to the processor. It is a device for staging the movement of data between main memory and processor registers to improve performance.

The three forms of memory just described are, typically, volatile and employ semiconductor technology. The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost. Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable magnetic disk, tape, and optical storage. External, nonvolatile memory is also referred to as **secondary memory** or **auxiliary memory**. These are used to store program and data files and are usually visible to the programmer only in terms of files and records, as opposed to individual bytes or words. Disk is also used to provide an extension to main memory known as virtual memory, which is discussed in Chapter 8.

Other forms of memory may be included in the hierarchy. For example, large IBM mainframes include a form of internal memory known as expanded storage. This uses a semiconductor technology that is slower and less expensive than that of main memory. Strictly speaking, this memory does not fit into the hierarchy but is a side branch: Data can be moved between main memory and expanded storage but not between expanded storage and external memory. Other forms of secondary memory include optical and magneto-optical disks. Finally, additional levels can be effectively added to the hierarchy in software. A portion of main memory can be used as a buffer to hold data temporarily that is to be read out to disk. Such a technique, sometimes referred to as a disk cache,[2] improves performance in two ways:

- Disk writes are clustered. Instead of many small transfers of data, we have a few large transfers of data. This improves disk performance and minimizes processor involvement.
- Some data destined for write-out may be referenced by a program before the next dump to disk. In that case, the data are retrieved rapidly from the software cache rather than slowly from the disk.

Appendix 4A examines the performance implications of multilevel memory structures.

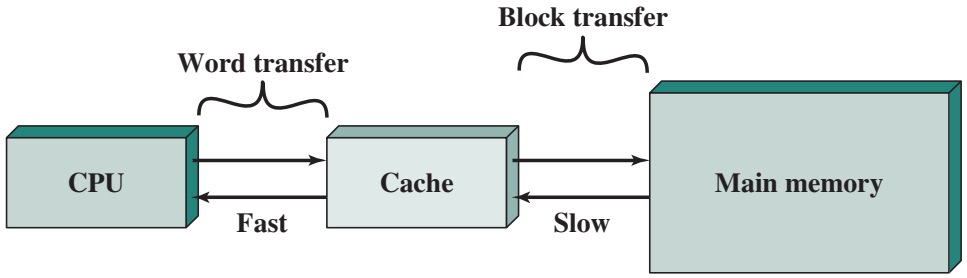
[2] Disk cache is generally a purely software technique and is not examined in this book. See [STAL15] for a discussion.

4.2 CACHE MEMORY PRINCIPLES

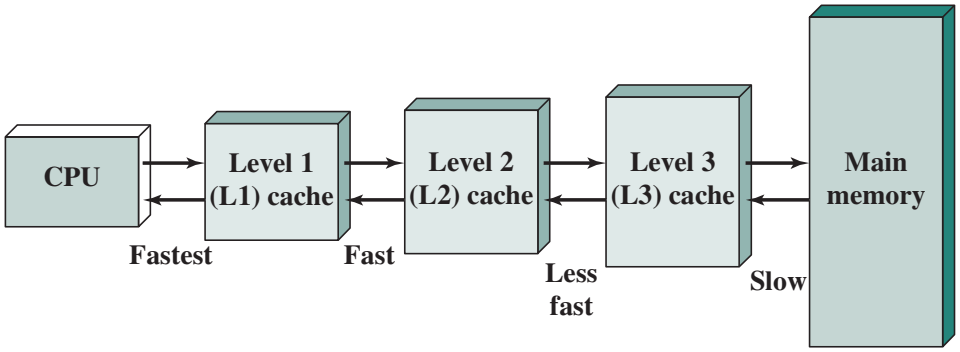
Cache memory is designed to combine the memory access time of expensive, high-speed memory combined with the large memory size of less expensive, lower-speed memory. The concept is illustrated in Figure 4.3a. There is a relatively large and slow main memory together with a smaller, faster cache memory. The cache contains a copy of portions of main memory. When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that there will be future references to that same memory location or to other words in the block.

Figure 4.3b depicts the use of multiple levels of cache. The L2 cache is slower and typically larger than the L1 cache, and the L3 cache is slower and typically larger than the L2 cache.

Figure 4.4 depicts the structure of a cache/main-memory system. Main memory consists of up to 2^n addressable words, with each word having a unique n-bit address. For mapping purposes, this memory is considered to consist of a number of fixed-length blocks of K words each. That is, there are $M = 2^n/K$ blocks in main memory. The cache consists of m blocks, called **lines**.[3] Each line contains K words,



(a) Single cache



(b) Three-level cache organization

Figure 4.3 Cache and Main Memory

[3]In referring to the basic unit of the cache, the term *line* is used, rather than the term *block*, for two reasons: (1) to avoid confusion with a main memory block, which contains the same number of data words as a cache line; and (2) because a cache line includes not only K words of data, just as a main memory block, but also includes tag and control bits.

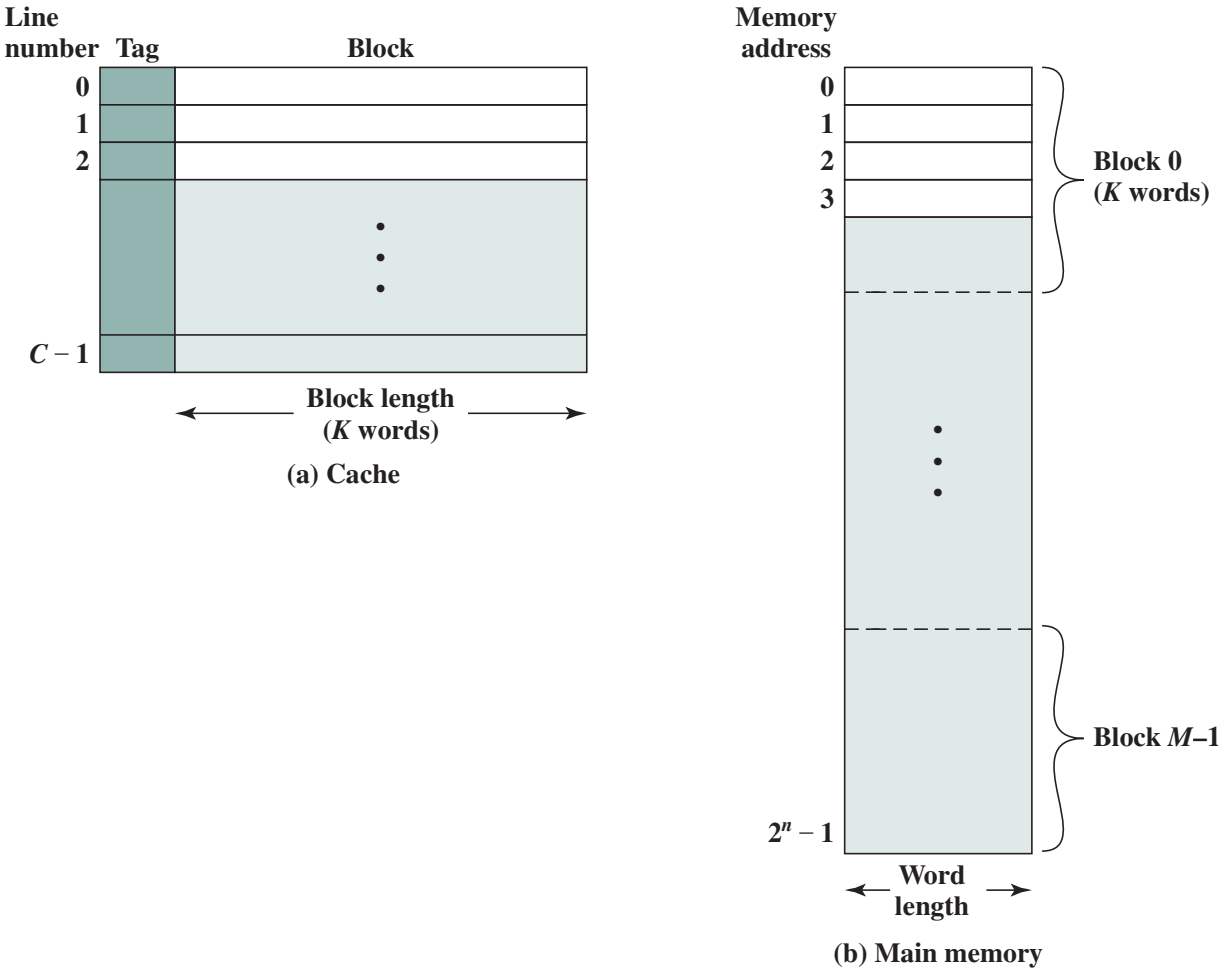


Figure 4.4 Cache/Main Memory Structure

plus a tag of a few bits. Each line also includes control bits (not shown), such as a bit to indicate whether the line has been modified since being loaded into the cache. The length of a line, not including tag and control bits, is the **line size**. The line size may be as small as 32 bits, with each "word" being a single byte; in this case the line size is 4 bytes. The number of lines is considerably less than the number of main memory blocks ($m \ll M$). At any time, some subset of the blocks of memory resides in lines in the cache. If a word in a block of memory is read, that block is transferred to one of the lines of the cache. Because there are more blocks than lines, an individual line cannot be uniquely and permanently dedicated to a particular block. Thus, each line includes a **tag** that identifies which particular block is currently being stored. The tag is usually a portion of the main memory address, as described later in this section.

Figure 4.5 illustrates the read operation. The processor generates the read address (RA) of a word to be read. If the word is contained in the cache, it is delivered to the processor. Otherwise, the block containing that word is loaded into the cache, and the word is delivered to the processor. Figure 4.5 shows these last two operations occurring in parallel and reflects the organization shown in Figure 4.6, which is typical of contemporary cache organizations. In this organization, the cache connects to the processor via data, control, and address lines. The data and address lines also attach to data and address buffers, which attach to a system bus from

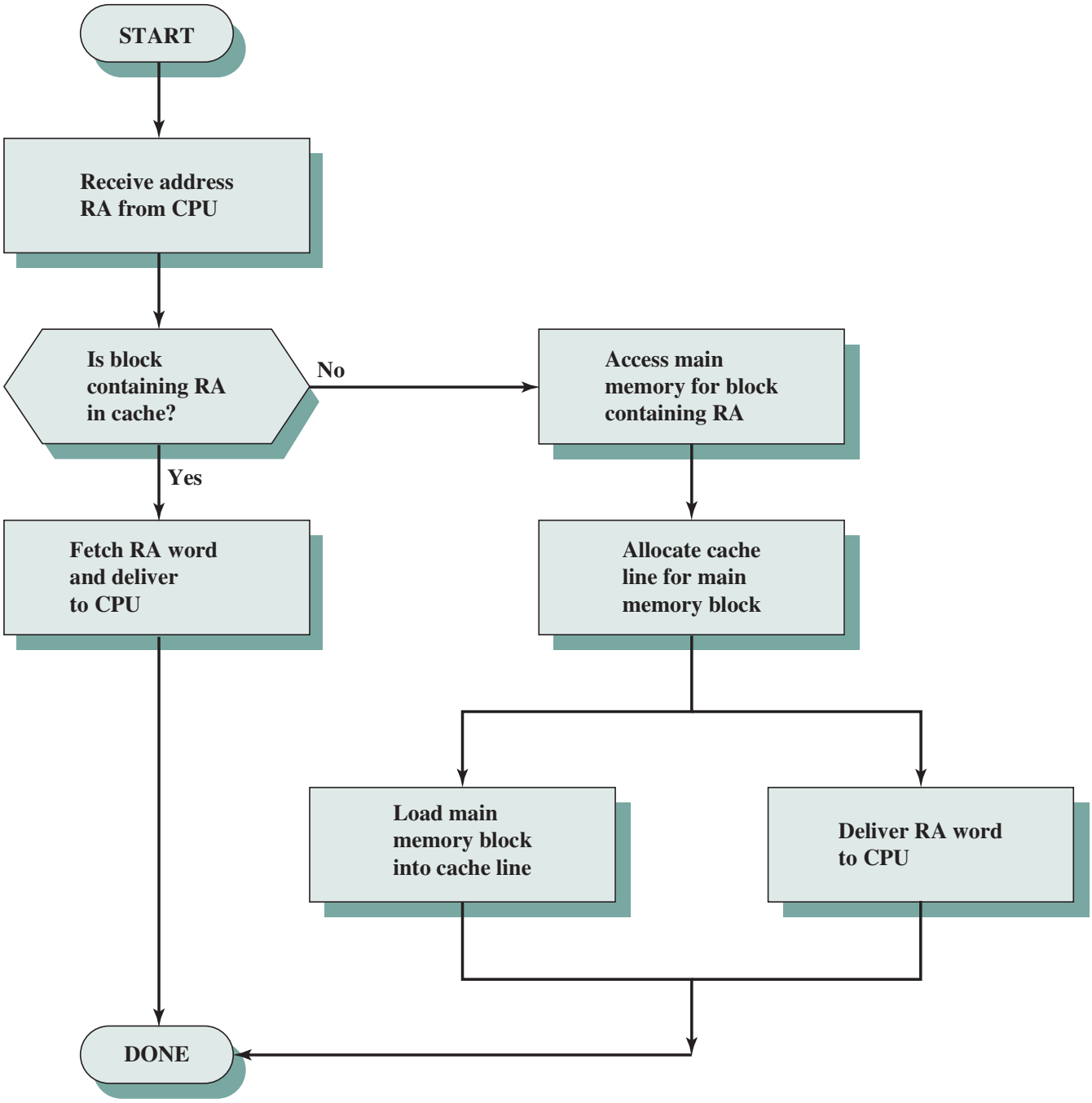


Figure 4.5 Cache Read Operation

which main memory is reached. When a cache hit occurs, the data and address buffers are disabled and communication is only between processor and cache, with no system bus traffic. When a cache miss occurs, the desired address is loaded onto the system bus and the data are returned through the data buffer to both the cache and the processor. In other organizations, the cache is physically interposed between the processor and the main memory for all data, address, and control lines. In this latter case, for a cache miss, the desired word is first read into the cache and then transferred from cache to processor.

A discussion of the performance parameters related to cache use is contained in Appendix 4A.

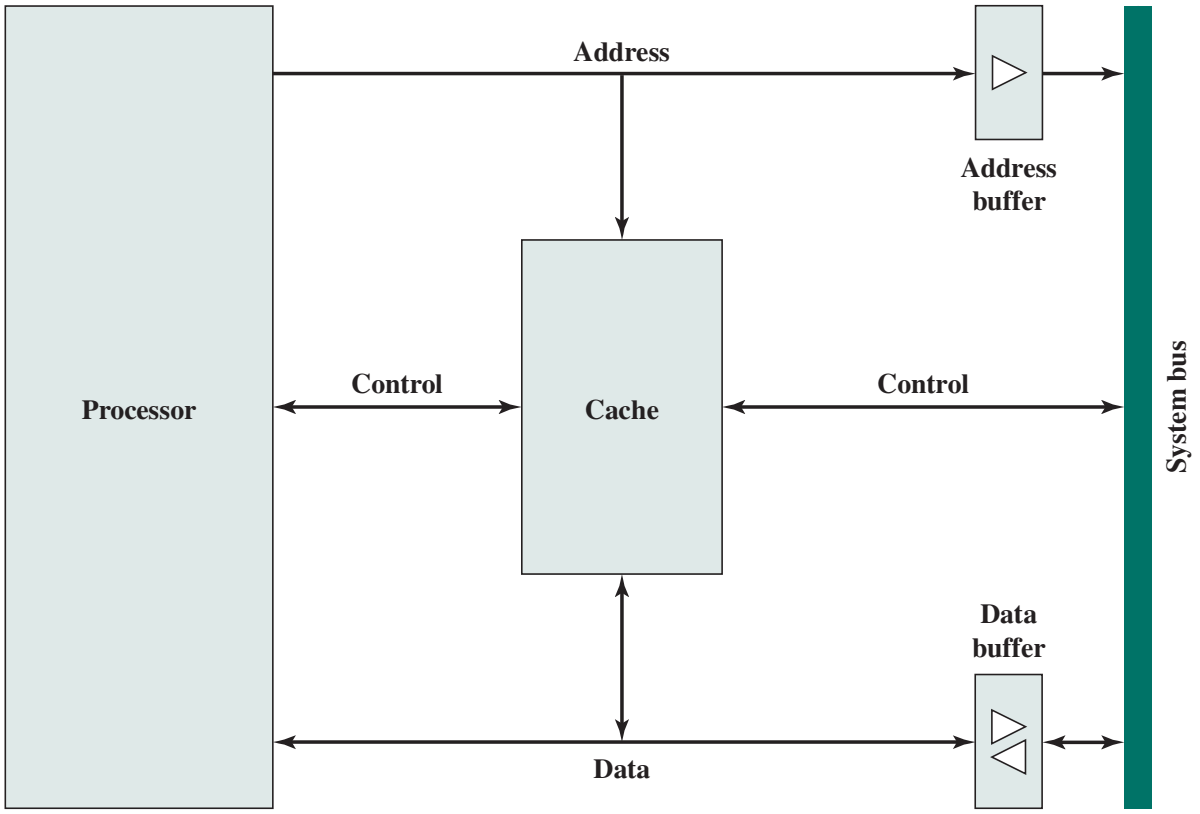


Figure 4.6 Typical Cache Organization

4.3 ELEMENTS OF CACHE DESIGN

This section provides an overview of cache design parameters and reports some typical results. We occasionally refer to the use of caches in **high-performance computing (HPC)**. HPC deals with supercomputers and their software, especially for scientific applications that involve large amounts of data, vector and matrix computation, and the use of parallel algorithms. Cache design for HPC is quite different than for other hardware platforms and applications. Indeed, many researchers have found that HPC applications perform poorly on computer architectures that employ caches [BAIL93]. Other researchers have since shown that a cache hierarchy can be useful in improving performance if the application software is tuned to exploit the cache [WANG99, PRES01].[4]

Although there are a large number of cache implementations, there are a few basic design elements that serve to classify and differentiate cache architectures. Table 4.2 lists key elements.

Cache Addresses

Almost all nonembedded processors, and many embedded processors, support virtual memory, a concept discussed in Chapter 8. In essence, virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available. When virtual memory is used, the address fields of machine instructions contain virtual addresses. For reads

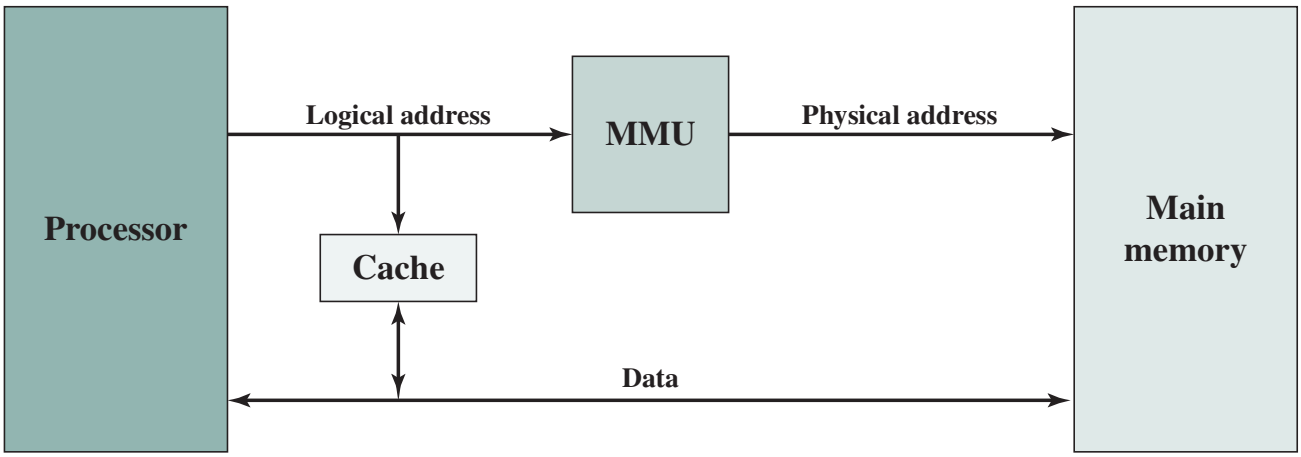
[4]For a general discussion of HPC, see [DOWD98].

Table 4.2 Elements of Cache Design

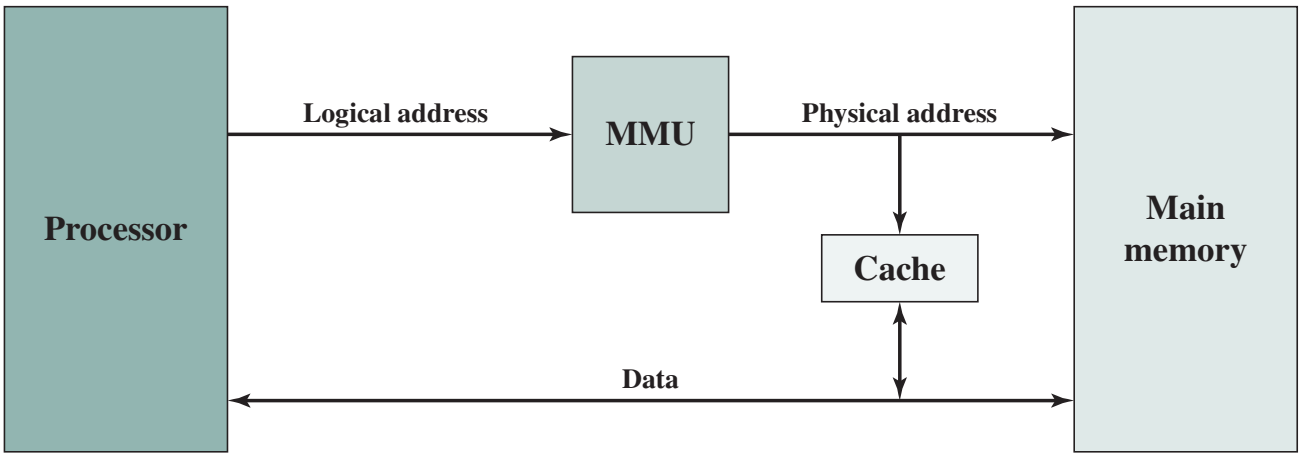
Cache Addresses	**Write Policy**
Logical	Write through
Physical	Write back
Cache Size	**Line Size**
Mapping Function	**Number of Caches**
Direct	Single or two level
Associative	Unified or split
Set associative	
Replacement Algorithm	
Least recently used (LRU)	
First in first out (FIFO)	
Least frequently used (LFU)	
Random	

to and writes from main memory, a hardware memory management unit (MMU) translates each virtual address into a physical address in main memory.

When virtual addresses are used, the system designer may choose to place the cache between the processor and the MMU or between the MMU and main memory (Figure 4.7). A **logical cache**, also known as a **virtual cache**, stores data using



(a) Logical cache



(b) Physical cache

Figure 4.7 Logical and Physical Caches

virtual addresses. The processor accesses the cache directly, without going through the MMU. A **physical cache** stores data using main memory **physical addresses**.

One obvious advantage of the logical cache is that cache access speed is faster than for a physical cache, because the cache can respond before the MMU performs an address translation. The disadvantage has to do with the fact that most virtual memory systems supply each application with the same virtual memory address space. That is, each application sees a virtual memory that starts at address 0. Thus, the same virtual address in two different applications refers to two different physical addresses. The cache memory must therefore be completely flushed with each application context switch, or extra bits must be added to each line of the cache to identify which virtual address space this address refers to.

The subject of logical versus physical cache is a complex one, and beyond the scope of this book. For a more in-depth discussion, see [CEKL97] and [JACO08].

Cache Size

The second item in Table 4.2, cache size, has already been discussed. We would like the size of the cache to be small enough so that the overall average cost per bit is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone. There are several other motivations for minimizing cache size. The larger the cache, the larger the number of gates involved in addressing the cache. The result is that large caches tend to be slightly slower than small ones—even when built with the same integrated circuit technology and put in the same place on chip and circuit board. The available chip and board area also limits cache size. Because the performance of the cache is very sensitive to the nature of the workload, it is impossible to arrive at a single "optimum" cache size. Table 4.3 lists the cache sizes of some current and past processors.

Mapping Function

Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines. Further, a means is needed for determining which main memory block currently occupies a cache line. The choice of the mapping function dictates how the cache is organized. Three techniques can be used: direct, associative, and set-associative. We examine each of these in turn. In each case, we look at the general structure and then a specific example.

EXAMPLE 4.2 For all three cases, the example includes the following elements:

- The cache can hold 64 kB.
- Data are transferred between main memory and the cache in blocks of 4 bytes each. This means that the cache is organized as 16K $= 2^{14}$ lines of 4 bytes each.
- The main memory consists of 16 MB, with each byte directly addressable by a 24-bit address ($2^{24} = 16$M). Thus, for mapping purposes, we can consider main memory to consist of 4M blocks of 4 bytes each.

Table 4.3 Cache Sizes of Some Processors

Processor	Type	Year of Introduction	L1 Cache[a]	L2 Cache	L3 Cache
IBM 360/85	Mainframe	1968	16–32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
VAX 11/780	Minicomputer	1978	16 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128–256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256–512 kB	—
PowerPC 601	PC	1993	32 kB	—	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
PowerPC G4	PC/server	1999	32 kB/32 kB	256 kB to 1 MB	2 MB
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 kB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—
CRAY MTA[b]	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 kB	4 MB
Itanium 2	PC/server	2002	32 kB	256 kB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1 MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24–48 MB
Intel Core i7 EE 990	Workstation/ server	2011	6 × 32 kB/ 32 kB	1.5 MB	12 MB
IBM zEnterprise 196	Mainframe/ server	2011	24 × 64 kB/ 128 kB	24 × 1.5 MB	24 MB L3 192 MB L4

Notes: [a] Two values separated by a slash refer to instruction and data caches. [b] Both caches are instruction only; no data caches.

DIRECT MAPPING The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. The mapping is expressed as

$$i = j \text{ modulo } m$$

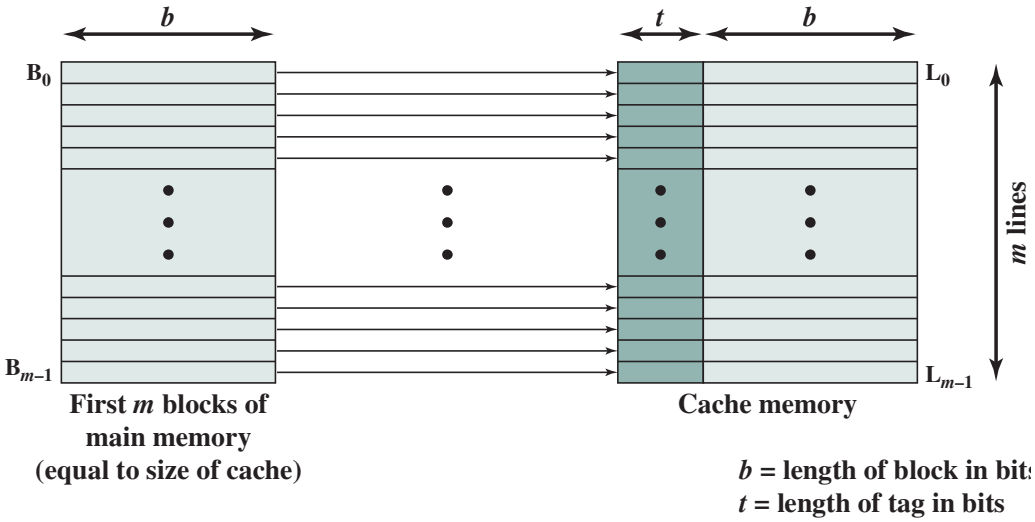
where

i = cache line number

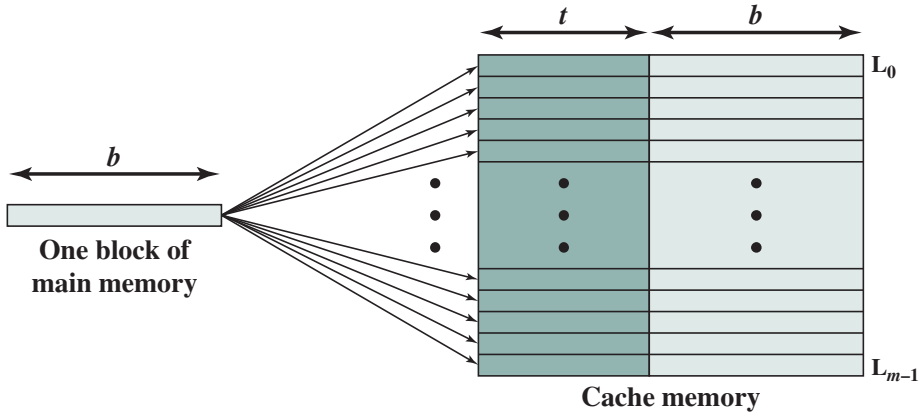
j = main memory block number

m = number of lines in the cache

Figure 4.8a shows the mapping for the first m blocks of main memory. Each block of main memory maps into one unique line of the cache. The next m blocks



(a) Direct mapping



(b) Associative mapping

Figure 4.8 Mapping from Main Memory to Cache: Direct and Associative

of main memory map into the cache in the same fashion; that is, block B_m of main memory maps into line L_0 of cache, block B_{m+1} maps into line L_1, and so on.

The mapping function is easily implemented using the main memory address. Figure 4.9 illustrates the general mechanism. For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory; in most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of $s - r$ bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m = 2^r$ lines of the cache. To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes
- Block size $=$ line size $= 2^w$ words or bytes

- Number of blocks in main memory $= \dfrac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache $= m = 2^r$
- Size of cache $= 2^{r+w}$ words or bytes
- Size of tag $= (s - r)$ bits

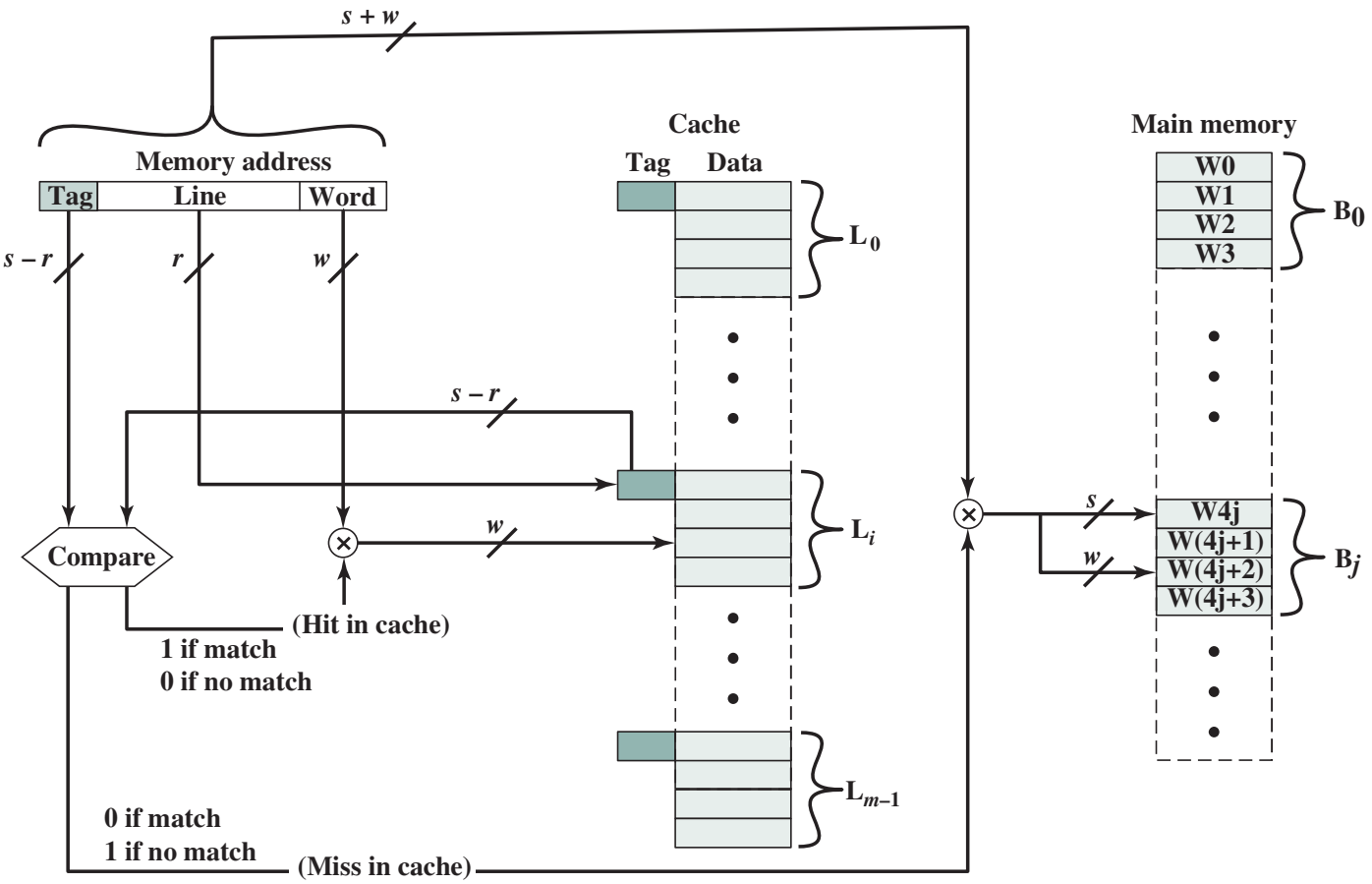


Figure 4.9 Direct-Mapping Cache Organization

EXAMPLE 4.2a Figure 4.10 shows our example system using direct mapping.[5] In the example, $m = 16\text{K} = 2^{14}$ and $i = j$ modulo 2^{14}. The mapping becomes

Cache Line	Starting Memory Address of Block
0	000000, 010000, …, FF0000
1	000004, 010004, …, FF0004
⋮	⋮
$2^{14} - 1$	00FFFC, 01FFFC, …, FFFFFC

Note that no two blocks that map into the same line number have the same tag number. Thus, blocks with starting addresses 000000, 010000, …, FF0000 have tag numbers 00, 01, …, FF, respectively.

Referring back to Figure 4.5, a read operation works as follows. The cache system is presented with a 24-bit address. The 14-bit line number is used as an index into the cache to access a particular line. If the 8-bit tag number matches the tag number currently stored in that line, then the 2-bit word number is used to select one of the 4 bytes in that line. Otherwise, the 22-bit tag-plus-line field is used to fetch a block from main memory. The actual address that is used for the fetch is the 22-bit tag-plus-line concatenated with two 0 bits, so that 4 bytes are fetched starting on a block boundary.

[5] In this and subsequent figures, memory values are represented in hexadecimal notation. See Chapter 9 for a basic refresher on number systems (decimal, binary, hexadecimal).

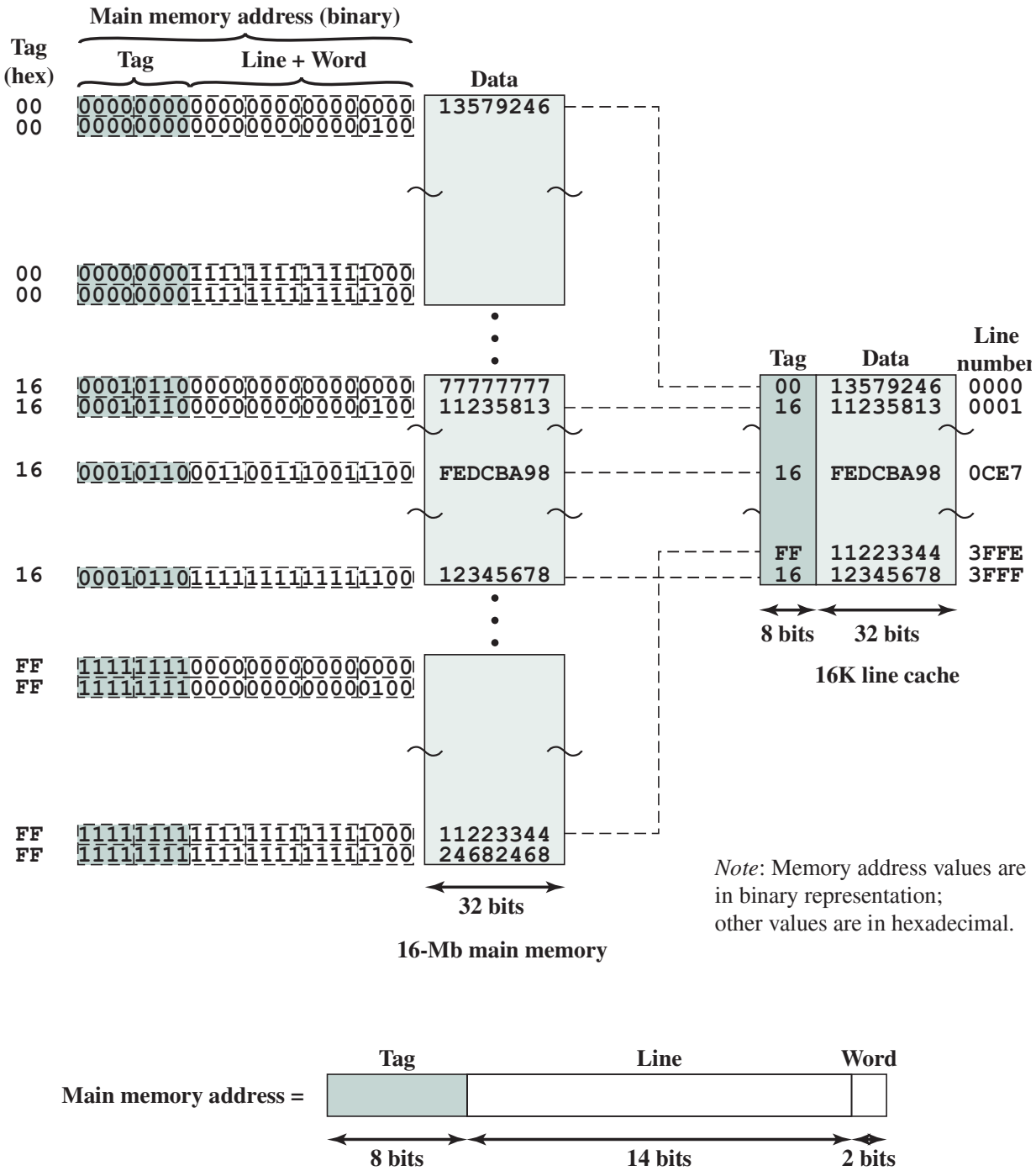


Figure 4.10 Direct Mapping Example

The effect of this mapping is that blocks of main memory are assigned to lines of the cache as follows:

Cache line	Main memory blocks assigned
0	$0, m, 2m, \ldots, 2^s - m$
1	$1, m + 1, 2m + 1, \ldots, 2^s - m + 1$
$\vdots$	$\vdots$
$m - 1$	$m - 1, 2m - 1, 3m - 1, \ldots, 2^s - 1$

Thus, the use of a portion of the address as a line number provides a unique mapping of each block of main memory into the cache. When a block is actually

read into its assigned line, it is necessary to tag the data to distinguish it from other blocks that can fit into that line. The most significant $s - r$ bits serve this purpose.

The direct mapping technique is simple and inexpensive to implement. Its main disadvantage is that there is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as *thrashing*).

Selective Victim Cache Simulator

One approach to lower the **miss** penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at a small cost. Such recycling is possible using a victim cache. Victim cache was originally proposed as an approach to reduce the conflict misses of direct mapped caches without affecting its fast access time. Victim cache is a fully associative cache, whose size is typically 4 to 16 cache lines, residing between a direct mapped L1 cache and the next level of memory. This concept is explored in Appendix F.

ASSOCIATIVE MAPPING Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache (Figure 4.8b). In this case, the cache control logic interprets a memory address simply as a Tag and a Word field. The Tag field uniquely identifies a block of main memory. To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's tag for a match. Figure 4.11 illustrates the logic.

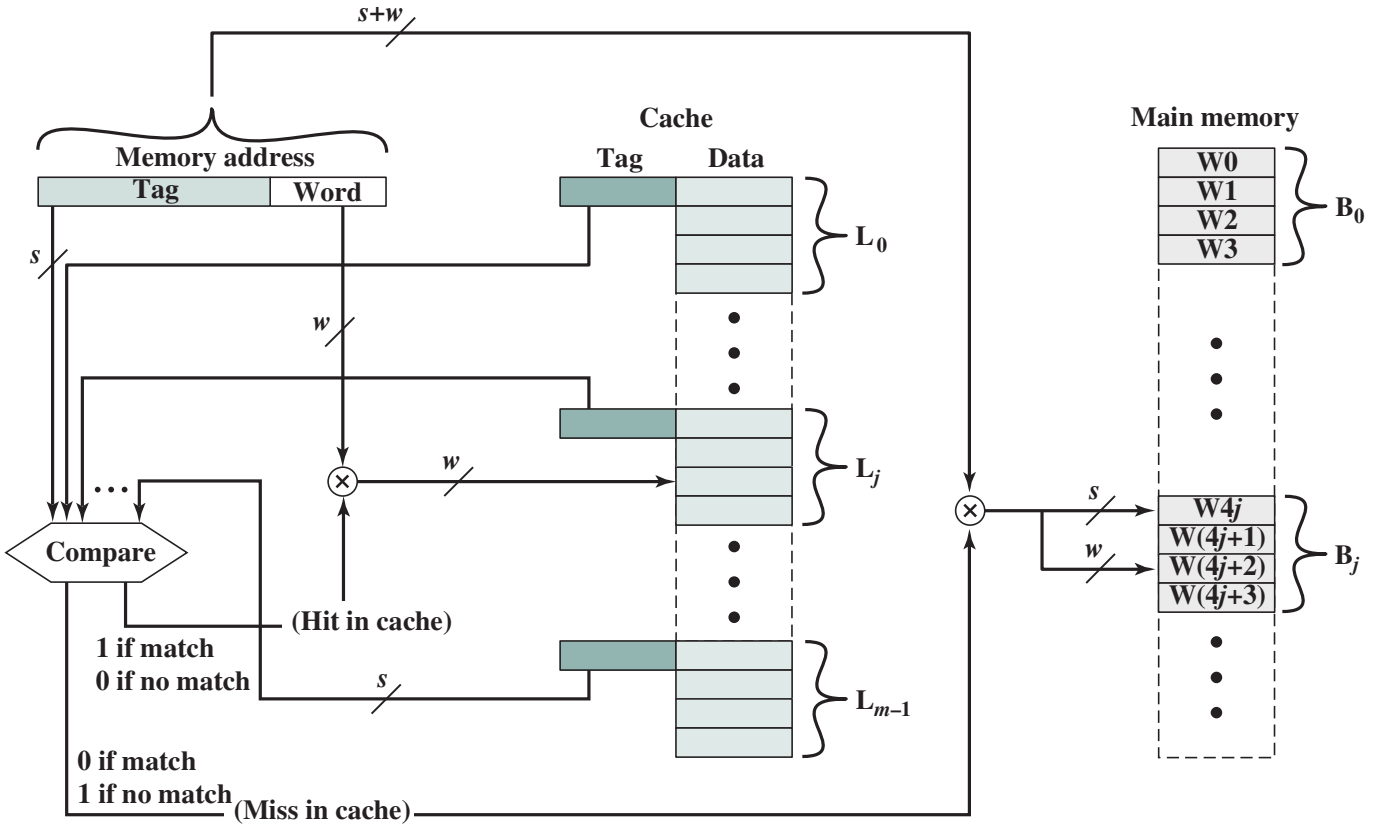


Figure 4.11 Fully Associative Cache Organization

EXAMPLE 4.2b Figure 4.12 shows our example using associative mapping. A main memory address consists of a 22-bit tag and a 2-bit byte number. The 22-bit tag must be stored with the 32-bit block of data for each line in the cache. Note that it is the leftmost (most significant) 22 bits of the address that form the tag. Thus, the 24-bit hexadecimal address 16339C has the 22-bit tag 058CE7. This is easily seen in binary notation:

Memory address	0001	0110	0011	0011	1001	1100	(binary)
	1	6	3	3	9	C	(hex)
Tag (leftmost 22 bits)	00	0101	1000	1100	1110	0111	(binary)
	0	5	8	C	E	7	(hex)

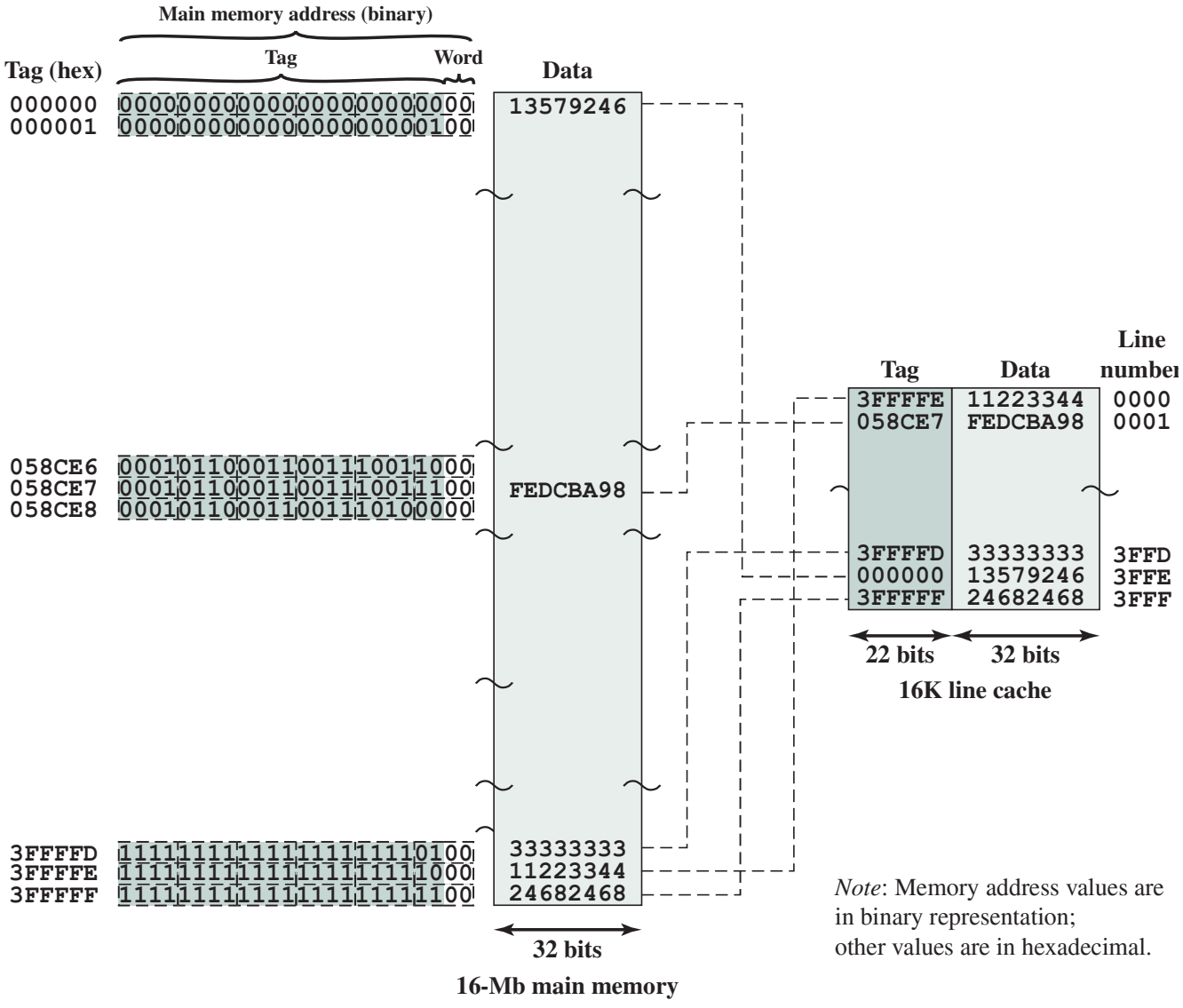


Figure 4.12 Associative Mapping Example

Note that no field in the address corresponds to the line number, so that the number of lines in the cache is not determined by the address format. To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes
- Block size = line size $= 2^w$ words or bytes
- Number of blocks in main memory $= \frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache = undetermined
- Size of tag = s bits

With associative mapping, there is flexibility as to which block to replace when a new block is read into the cache. Replacement algorithms, discussed later in this section, are designed to maximize the hit ratio. The principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel.

Cache Time Analysis Simulator

SET-ASSOCIATIVE MAPPING Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages.

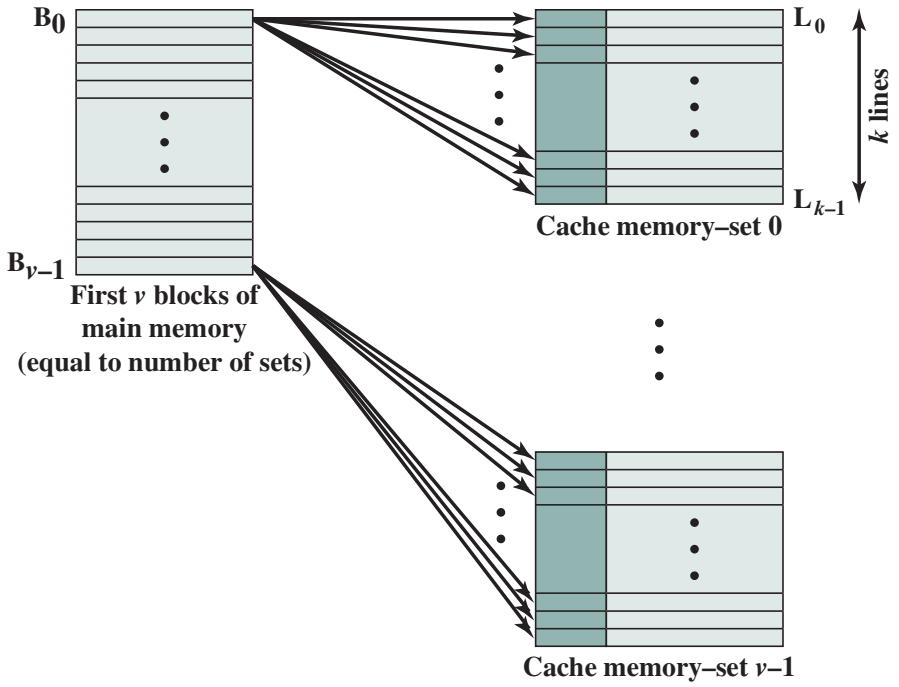
In this case, the cache consists of number sets, each of which consists of a number of lines. The relationships are

$$m = v \times k$$
$$i = j \text{ modulo } v$$

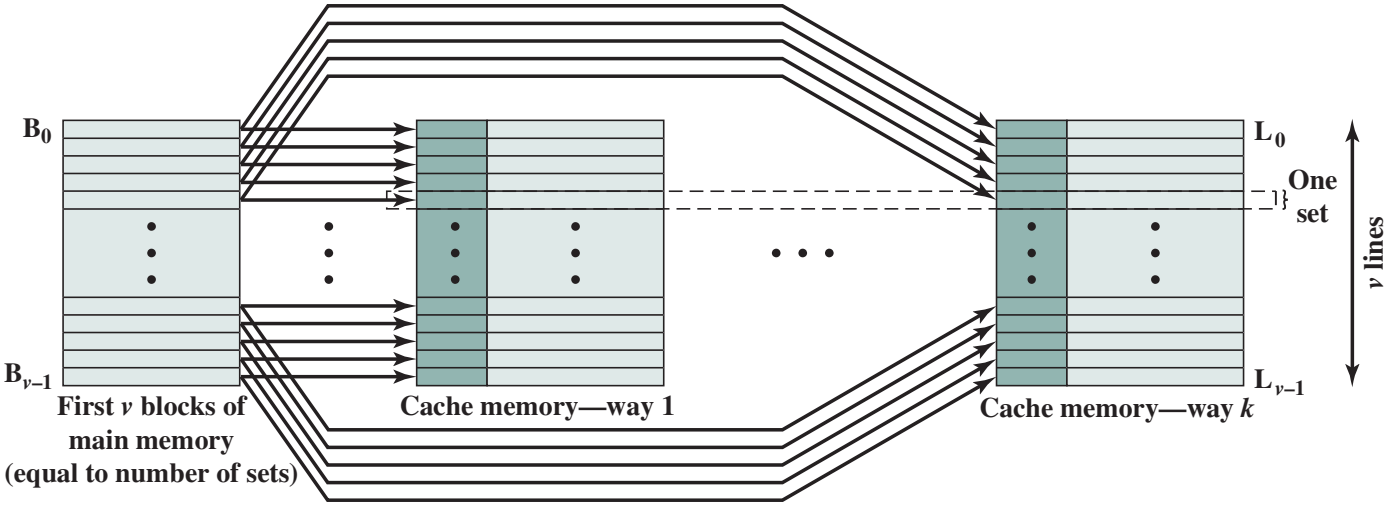
where

i = cache set number
j = main memory block number
m = number of lines in the cache
v = number of sets
k = number of lines in each set

This is referred to as k-way set-associative mapping. With set-associative mapping, block B_j can be mapped into any of the lines of set j. Figure 4.13a illustrates this mapping for the first v blocks of main memory. As with associative mapping, each word maps into multiple cache lines. For set-associative mapping, each word maps into all the cache lines in a specific set, so that main memory block B_0 maps into set 0, and so on. Thus, the set-associative cache can be physically implemented as v associative caches. It is also possible to implement the set-associative cache as k direct mapping caches, as shown in Figure 4.13b. Each direct-mapped cache is referred to as a *way,* consisting of v lines. The first v lines of main memory are direct mapped into the v lines of each way; the next group of v lines of main memory are similarly mapped, and so on. The direct-mapped implementation is typically used



(a) *v* associative–mapped caches



(b) *k* direct–mapped caches

Figure 4.13 Mapping from Main Memory to Cache: *k*-Way Set Associative

for small degrees of associativity (small values of k) while the associative-mapped implementation is typically used for higher degrees of associativity [JACO08].

For set-associative mapping, the cache control logic interprets a memory address as three fields: Tag, Set, and Word. The d set bits specify one of $v = 2^d$ sets. The s bits of the Tag and Set fields specify one of the 2^s blocks of main memory. Figure 4.14 illustrates the cache control logic. With fully associative mapping, the tag in a memory address is quite large and must be compared to the tag of every line in the cache. With k-way set-associative mapping, the tag in a memory address is much smaller and is only compared to the k tags within a single set. To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes

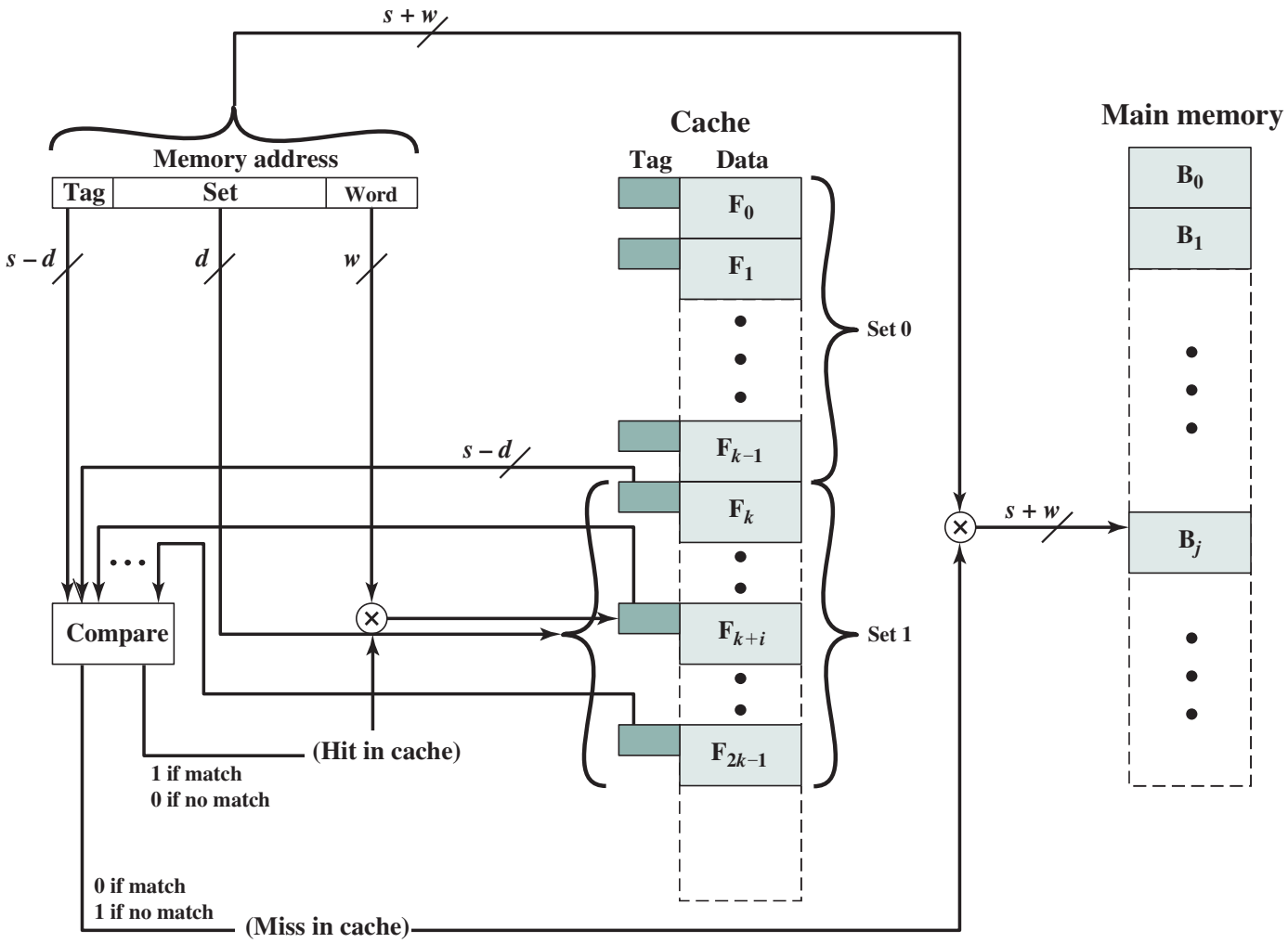


Figure 4.14 *k*-Way Set-Associative Cache Organization

- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in set = k
- Number of sets = $v = 2^d$
- Number of lines in cache = $m = kv = k \times 2^d$
- Size of cache = $k \times 2^{d+w}$ words or bytes
- Size of tag = $(s - d)$ bits

EXAMPLE 4.2c Figure 4.15 shows our example using set-associative mapping with two lines in each set, referred to as two-way set-associative. The 13-bit set number identifies a unique set of two lines within the cache. It also gives the number of the block in main memory, modulo 2^{13}. This determines the mapping of blocks into lines. Thus, blocks 000000, 008000, …, FF8000 of main memory map into cache set 0. Any of those blocks can be loaded into either of the two lines in the set. Note that no two blocks that map into the same cache set have the same tag number. For a read operation, the 13-bit set number is used to determine which set of two lines is to be examined. Both lines in the set are examined for a match with the tag number of the address to be accessed.

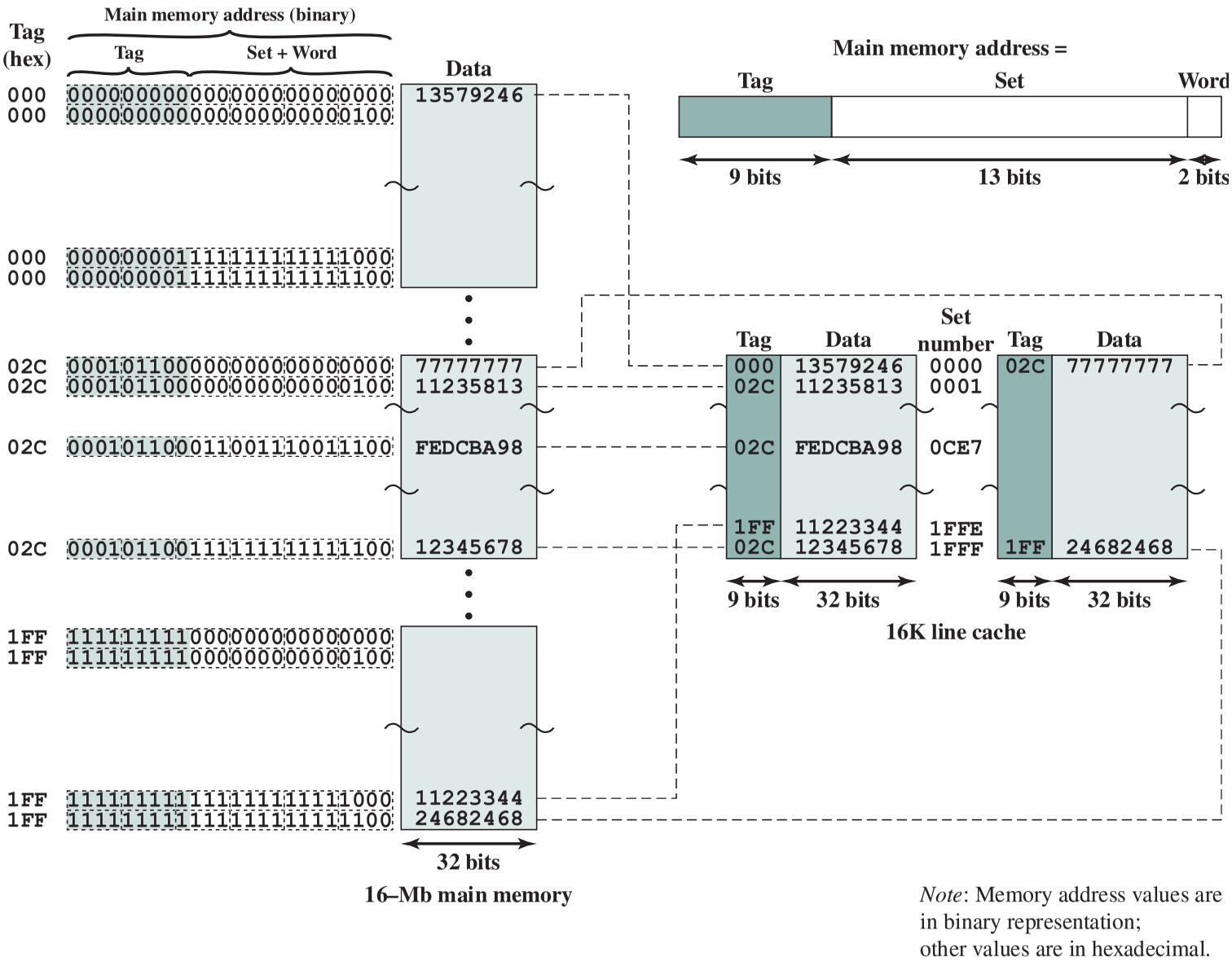


Figure 4.15 Two-Way Set-Associative Mapping Example

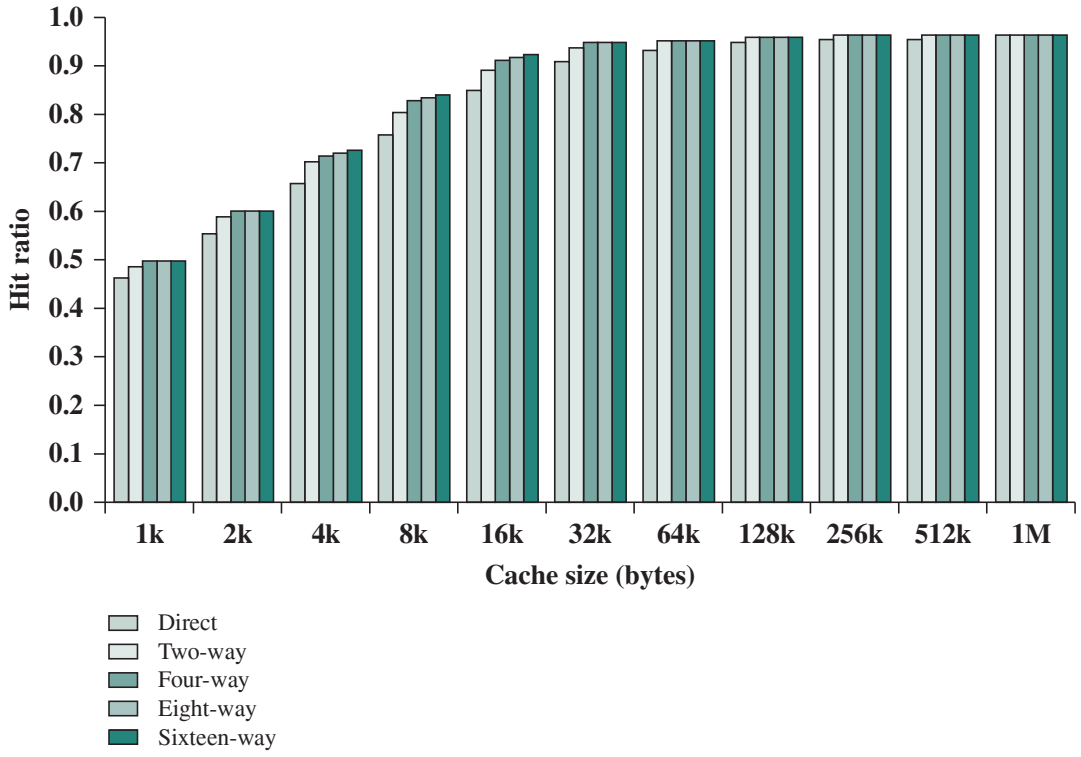


Figure 4.16 Varying Associativity over Cache Size

In the extreme case of $v = m, k = 1$, the set-associative technique reduces to direct mapping, and for $v = 1, k = m$, it reduces to associative mapping. The use of two lines per set ($v = m/2, k = 2$) is the most common set-associative organization. It significantly improves the hit ratio over direct mapping. Four-way set associative ($v = m/4, k = 4$) makes a modest additional improvement for a relatively small additional cost [MAYB84, HILL89]. Further increases in the number of lines per set have little effect.

Figure 4.16 shows the results of one simulation study of set-associative cache performance as a function of cache size [GENU04]. The difference in performance between direct and two-way set associative is significant up to at least a cache size of 64 kB. Note also that the difference between two-way and four-way at 4 kB is much less than the difference in going from for 4 kB to 8 kB in cache size. The complexity of the cache increases in proportion to the associativity, and in this case would not be justifiable against increasing cache size to 8 or even 16 kB. A final point to note is that beyond about 32 kB, increase in cache size brings no significant increase in performance.

The results of Figure 4.16 are based on simulating the execution of a GCC compiler. Different applications may yield different results. For example, [CANT01] reports on the results for cache performance using many of the CPU2000 SPEC benchmarks. The results of [CANT01] in comparing hit ratio to cache size follow the same pattern as Figure 4.16, but the specific values are somewhat different.

Cache Simulator
Multitask Cache Simulator

Replacement Algorithms

Once the cache has been filled, when a new block is brought into the cache, one of the existing blocks must be replaced. For direct mapping, there is only one possible line for any particular block, and no choice is possible. For the associative and set-associative techniques, a replacement algorithm is needed. To achieve high speed, such an algorithm must be implemented in hardware. A number of algorithms have been tried. We mention four of the most common. Probably the most effective is **least recently used (LRU)**: Replace that block in the set that has been in the cache longest with no reference to it. For two-way set associative, this is easily implemented. Each line includes a USE bit. When a line is referenced, its USE bit is set to 1 and the USE bit of the other line in that set is set to 0. When a block is to be read into the set, the line whose USE bit is 0 is used. Because we are assuming that more recently used memory locations are more likely to be referenced, LRU should give the best hit ratio. LRU is also relatively easy to implement for a fully associative cache. The cache mechanism maintains a separate list of indexes to all the lines in the cache. When a line is referenced, it moves to the front of the list. For replacement, the line at the back of the list is used. Because of its simplicity of implementation, LRU is the most popular replacement algorithm.

Another possibility is first-in-first-out (FIFO): Replace that block in the set that has been in the cache longest. FIFO is easily implemented as a round-robin or circular buffer technique. Still another possibility is least frequently used (LFU): Replace that block in the set that has experienced the fewest references. LFU could be implemented by associating a counter with each line. A technique not based on usage (i.e., not LRU, LFU, FIFO, or some variant) is to pick a line at random from among the candidate lines. Simulation studies have shown that random replacement provides only slightly inferior performance to an algorithm based on usage [SMIT82].

Write Policy

When a block that is resident in the cache is to be replaced, there are two cases to consider. If the old block in the cache has not been altered, then it may be overwritten with a new block without first writing out the old block. If at least one write operation has been performed on a word in that line of the cache, then main memory must be updated by writing the line of cache out to the block of memory before bringing in the new block. A variety of write policies, with performance and economic trade-offs, is possible. There are two problems to contend with. First, more than one device may have access to main memory. For example, an I/O module may be able to read-write directly to memory. If a word has been altered only in the cache, then the corresponding memory word is invalid. Further, if the I/O device has altered main memory, then the cache word is invalid. A more complex problem occurs when multiple processors are attached to the same bus and each processor has its own local cache. Then, if a word is altered in one cache, it could conceivably invalidate a word in other caches.

The simplest technique is called **write through**. Using this technique, all write operations are made to main memory as well as to the cache, ensuring that main memory is always valid. Any other processor–cache module can monitor traffic to main memory to maintain consistency within its own cache. The main disadvantage

of this technique is that it generates substantial memory traffic and may create a bottleneck. An alternative technique, known as **write back**, minimizes memory writes. With write back, updates are made only in the cache. When an update occurs, a **dirty bit**, or **use bit**, associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set. The problem with write back is that portions of main memory are invalid, and hence accesses by I/O modules can be allowed only through the cache. This makes for complex circuitry and a potential bottleneck. Experience has shown that the percentage of memory references that are writes is on the order of 15% [SMIT82]. However, for HPC applications, this number may approach 33% (vector-vector multiplication) and can go as high as 50% (matrix transposition).

EXAMPLE 4.3 Consider a cache with a line size of 32 bytes and a main memory that requires 30 ns to transfer a 4-byte word. For any line that is written at least once before being swapped out of the cache, what is the average number of times that the line must be written before being swapped out for a write-back cache to be more efficient that a write-through cache?

For the write-back case, each dirty line is written back once, at swap-out time, taking $8 \times 30 = 240$ ns. For the write-through case, each update of the line requires that one word be written out to main memory, taking 30 ns. Therefore, if the average line that gets written at least once gets written more than 8 times before swap out, then write back is more efficient.

In a bus organization in which more than one device (typically a processor) has a cache and main memory is shared, a new problem is introduced. If data in one cache are altered, this invalidates not only the corresponding word in main memory, but also that same word in other caches (if any other cache happens to have that same word). Even if a write-through policy is used, the other caches may contain invalid data. A system that prevents this problem is said to maintain cache coherency. Possible approaches to cache coherency include the following:

- **Bus watching with write through:** Each cache controller monitors the address lines to detect write operations to memory by other bus masters. If another master writes to a location in shared memory that also resides in the cache memory, the cache controller invalidates that cache entry. This strategy depends on the use of a write-through policy by all cache controllers.
- **Hardware transparency:** Additional hardware is used to ensure that all updates to main memory via cache are reflected in all caches. Thus, if one processor modifies a word in its cache, this update is written to main memory. In addition, any matching words in other caches are similarly updated.
- **Noncacheable memory:** Only a portion of main memory is shared by more than one processor, and this is designated as noncacheable. In such a system, all accesses to shared memory are cache misses, because the shared memory is never copied into the cache. The noncacheable memory can be identified using chip-select logic or high-address bits.

Cache coherency is an active field of research. This topic is explored further in Part Five.

Line Size

Another design element is the line size. When a block of data is retrieved and placed in the cache, not only the desired word but also some number of adjacent words are retrieved. As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality, which states that data in the vicinity of a referenced word are likely to be referenced in the near future. As the block size increases, more useful data are brought into the cache. The hit ratio will begin to decrease, however, as the block becomes even bigger and the probability of using the newly fetched information becomes less than the probability of reusing the information that has to be replaced. Two specific effects come into play:

- Larger blocks reduce the number of blocks that fit into a cache. Because each block fetch overwrites older cache contents, a small number of blocks results in data being overwritten shortly after they are fetched.
- As a block becomes larger, each additional word is farther from the requested word and therefore less likely to be needed in the near future.

The relationship between block size and hit ratio is complex, depending on the locality characteristics of a particular program, and no definitive optimum value has been found. A size of from 8 to 64 bytes seems reasonably close to optimum [SMIT87, PRZY88, PRZY90, HAND98]. For HPC systems, 64- and 128-byte cache line sizes are most frequently used.

Number of Caches

When caches were originally introduced, the typical system had a single cache. More recently, the use of multiple caches has become the norm. Two aspects of this design issue concern the number of levels of caches and the use of unified versus split caches.

MULTILEVEL CACHES As logic density has increased, it has become possible to have a cache on the same chip as the processor: the on-chip cache. Compared with a cache reachable via an external bus, the on-chip cache reduces the processor's external bus activity and therefore speeds up execution times and increases overall system performance. When the requested instruction or data is found in the on-chip cache, the bus access is eliminated. Because of the short data paths internal to the processor, compared with bus lengths, on-chip cache accesses will complete appreciably faster than would even zero-wait state bus cycles. Furthermore, during this period the bus is free to support other transfers.

The inclusion of an on-chip cache leaves open the question of whether an off-chip, or external, cache is still desirable. Typically, the answer is yes, and most contemporary designs include both on-chip and external caches. The simplest such organization is known as a two-level cache, with the internal level 1 (L1) and the external cache designated as level 2 (L2). The reason for including an L2 cache is the following: If there is no L2 cache and the processor makes an access request for a memory location not in the L1 cache, then the processor must access DRAM or

ROM memory across the bus. Due to the typically slow bus speed and slow memory access time, this results in poor performance. On the other hand, if an L2 SRAM (static RAM) cache is used, then frequently the missing information can be quickly retrieved. If the SRAM is fast enough to match the bus speed, then the data can be accessed using a zero-wait state transaction, the fastest type of bus transfer.

Two features of contemporary cache design for multilevel caches are noteworthy. First, for an off-chip L2 cache, many designs do not use the system bus as the path for transfer between the L2 cache and the processor, but use a separate data path, so as to reduce the burden on the system bus. Second, with the continued shrinkage of processor components, a number of processors now incorporate the L2 cache on the processor chip, improving performance.

The potential savings due to the use of an L2 cache depends on the hit rates in both the L1 and L2 caches. Several studies have shown that, in general, the use of a second-level cache does improve performance (e.g., see [AZIM92], [NOVI93], [HAND98]). However, the use of multilevel caches does complicate all of the design issues related to caches, including size, replacement algorithm, and write policy; see [HAND98] and [PEIR99] for discussions.

Figure 4.17 shows the results of one simulation study of two-level cache performance as a function of cache size [GENU04]. The figure assumes that both caches have the same line size and shows the total hit ratio. That is, a hit is counted if the desired data appears in either the L1 or the L2 cache. The figure shows the impact of L2 on total hits with respect to L1 size. L2 has little effect on the total number of cache hits until it is at least double the L1 cache size. Note that the steepest part of the slope for an L1 cache of 8 kB is for an L2 cache of 16 kB. Again for an L1 cache of 16 kB, the steepest part of the curve is for an L2 cache size of 32 kB. Prior to that point, the L2 cache has little, if any, impact on total cache performance. The need for the L2 cache to be larger than

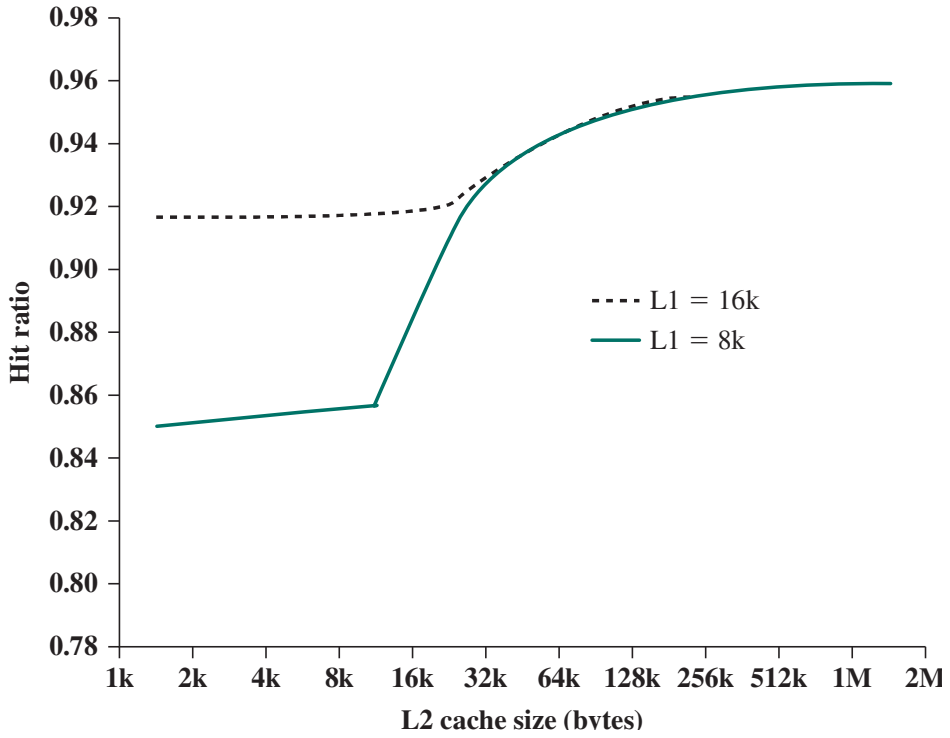


Figure 4.17 Total Hit Ratio (L1 and L2) for 8-kB and 16-kB L1

the L1 cache to affect performance makes sense. If the L2 cache has the same line size and capacity as the L1 cache, its contents will more or less mirror those of the L1 cache.

With the increasing availability of on-chip area available for cache, most contemporary microprocessors have moved the L2 cache onto the processor chip and added an L3 cache. Originally, the L3 cache was accessible over the external bus. More recently, most microprocessors have incorporated an on-chip L3 cache. In either case, there appears to be a performance advantage to adding the third level (e.g., see [GHAI98]). Further, large systems, such as the IBM mainframe zEnterprise systems, now incorporate 3 on-chip cache levels and a fourth level of cache shared across multiple chips [CURR11].

UNIFIED VERSUS SPLIT CACHES When the on-chip cache first made an appearance, many of the designs consisted of a single cache used to store references to both data and instructions. More recently, it has become common to split the cache into two: one dedicated to instructions and one dedicated to data. These two caches both exist at the same level, typically as two L1 caches. When the processor attempts to fetch an instruction from main memory, it first consults the instruction L1 cache, and when the processor attempts to fetch data from main memory, it first consults the data L1 cache.

There are two potential advantages of a unified cache:

- For a given cache size, a unified cache has a higher hit rate than split caches because it balances the load between instruction and data fetches automatically. That is, if an execution pattern involves many more instruction fetches than data fetches, then the cache will tend to fill up with instructions, and if an execution pattern involves relatively more data fetches, the opposite will occur.
- Only one cache needs to be designed and implemented.

The trend is toward split caches at the L1 and unified caches for higher levels, particularly for superscalar machines, which emphasize parallel instruction execution and the prefetching of predicted future instructions. The key advantage of the split cache design is that it eliminates contention for the cache between the instruction fetch/decode unit and the execution unit. This is important in any design that relies on the pipelining of instructions. Typically, the processor will fetch instructions ahead of time and fill a buffer, or pipeline, with instructions to be executed. Suppose now that we have a unified instruction/data cache. When the execution unit performs a memory access to load and store data, the request is submitted to the unified cache. If, at the same time, the instruction prefetcher issues a read request to the cache for an instruction, that request will be temporarily blocked so that the cache can service the execution unit first, enabling it to complete the currently executing instruction. This cache contention can degrade performance by interfering with efficient use of the instruction pipeline. The split cache structure overcomes this difficulty.

4.4 PENTIUM 4 CACHE ORGANIZATION

The evolution of cache organization is seen clearly in the evolution of Intel microprocessors (Table 4.4). The 80386 does not include an on-chip cache. The 80486 includes a single on-chip cache of 8 kB, using a line size of 16 bytes and a four-way

Table 4.4 Intel Cache Evolution

Problem	Solution	Processor on Which Feature First Appears
External memory slower than the system bus.	Add external cache using faster memory technology.	386
Increased processor speed results in external bus becoming a bottleneck for cache access.	Move external cache on-chip, operating at the same speed as the processor.	486
Internal cache is rather small, due to limited space on chip.	Add external L2 cache using faster technology than main memory.	486
Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit's data access takes place.	Create separate data and instruction caches.	Pentium
Increased processor speed results in external bus becoming a bottleneck for L2 cache access.	Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache.	Pentium Pro
	Move L2 cache on to the processor chip.	Pentium II
Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small.	Add external L3 cache.	Pentium III
	Move L3 cache on-chip.	Pentium 4

set-associative organization. All of the Pentium processors include two on-chip L1 caches, one for data and one for instructions. For the Pentium 4, the L1 data cache is 16 kB, using a line size of 64 bytes and a four-way set-associative organization. The Pentium 4 instruction cache is described subsequently. The Pentium II also includes an L2 cache that feeds both of the L1 caches. The L2 cache is eight-way set associative with a size of 512 kB and a line size of 128 bytes. An L3 cache was added for the Pentium III and became on-chip with high-end versions of the Pentium 4.

Figure 4.18 provides a simplified view of the Pentium 4 organization, highlighting the placement of the three caches. The processor core consists of four major components:

- **Fetch/decode unit:** Fetches program instructions in order from the L2 cache, decodes these into a series of micro-operations, and stores the results in the L1 instruction cache.
- **Out-of-order execution logic:** Schedules execution of the micro-operations subject to data dependencies and resource availability; thus, micro-operations may be scheduled for execution in a different order than they were fetched from the instruction stream. As time permits, this unit schedules speculative execution of micro-operations that may be required in the future.

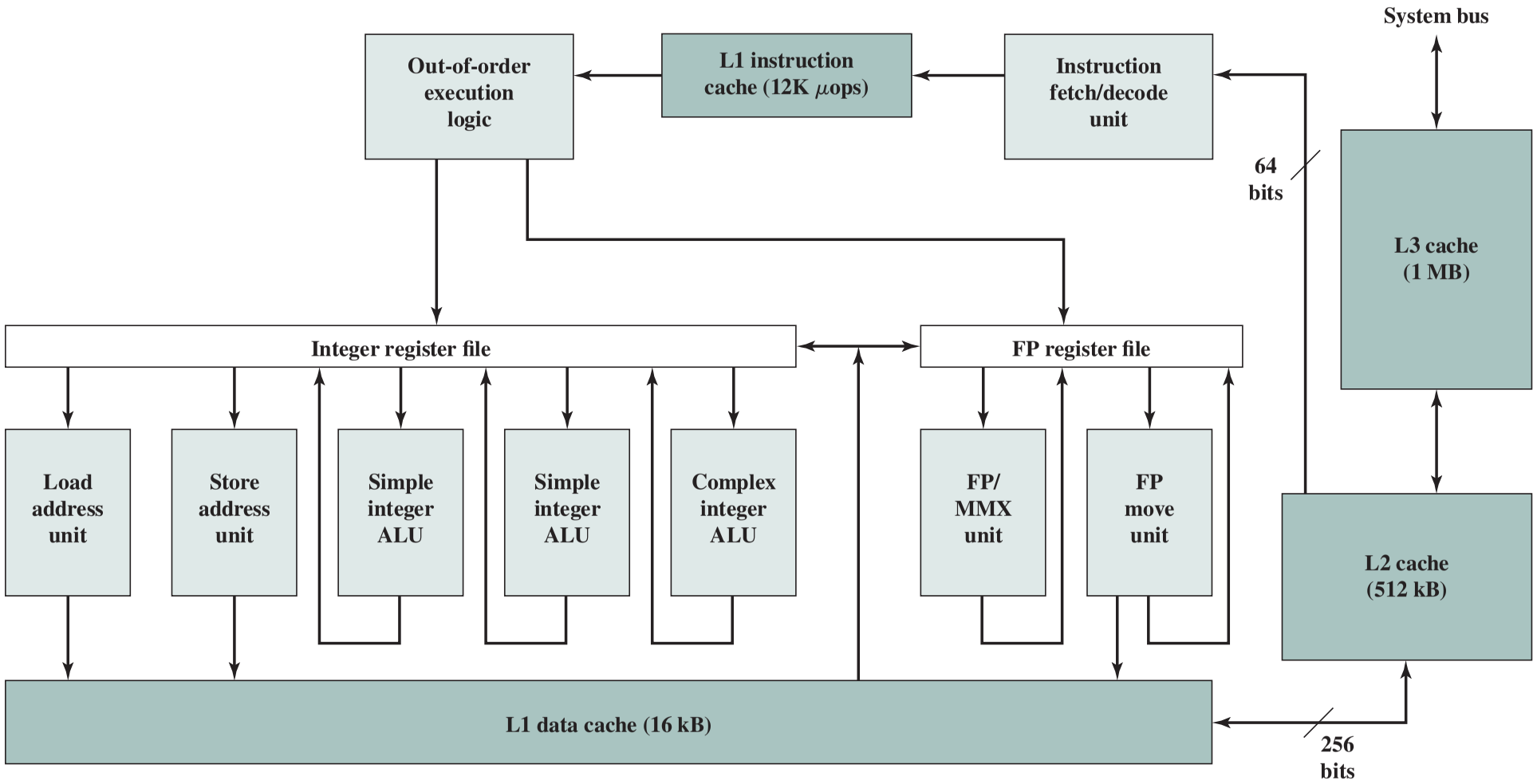


Figure 4.18 Pentium 4 Block Diagram

Table 4.5 Pentium 4 Cache Operating Modes

Control Bits		Operating Mode		
CD	NW	Cache Fills	Write Throughs	Invalidates
0	0	Enabled	Enabled	Enabled
1	0	Disabled	Enabled	Enabled
1	1	Disabled	Disabled	Disabled

Note: CD = 0; NW = 1 is an invalid combination.

- **Execution units:** These units execute micro-operations, fetching the required data from the L1 data cache and temporarily storing results in registers.
- **Memory subsystem:** This unit includes the L2 and L3 caches and the system bus, which is used to access main memory when the L1 and L2 caches have a cache miss and to access the system I/O resources.

Unlike the organization used in all previous Pentium models, and in most other processors, the Pentium 4 instruction cache sits between the instruction decode logic and the execution core. The reasoning behind this design decision is as follows: As discussed more fully in Chapter 16, the Pentium process decodes, or translates, Pentium machine instructions into simple RISC-like instructions called micro-operations. The use of simple, fixed-length micro-operations enables the use of superscalar pipelining and scheduling techniques that enhance performance. However, the Pentium machine instructions are cumbersome to decode; they have a variable number of bytes and many different options. It turns out that performance is enhanced if this decoding is done independently of the scheduling and pipelining logic. We return to this topic in Chapter 16.

The data cache employs a write-back policy: Data are written to main memory only when they are removed from the cache and there has been an update. The Pentium 4 processor can be dynamically configured to support write-through caching.

The L1 data cache is controlled by two bits in one of the control registers, labeled the CD (cache disable) and NW (not write-through) bits (Table 4.5). There are also two Pentium 4 instructions that can be used to control the data cache: INVD invalidates (flushes) the internal cache memory and signals the external cache (if any) to invalidate. WBINVD writes back and invalidates internal cache and then writes back and invalidates external cache.

Both the L2 and L3 caches are eight-way set-associative with a line size of 128 bytes.

4.5 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

access time	cache line	cache set
associative mapping	cache memory	data cache
cache hit	cache miss	direct access

direct mapping	logical cache	spatial locality
high-performance computing (HPC)	memory hierarchy	split cache
hit	miss	tag
hit ratio	multilevel cache	temporal locality
instruction cache	physical address	unified cache
L1 cache	physical cache	virtual address
L2 cache	random access	virtual cache
L3 cache	replacement algorithm	write back
line	secondary memory	write through
locality	sequential access	
	set-associative mapping	

Review Questions

4.1 What are the differences among sequential access, direct access, and random access?

4.2 What is the general relationship among access time, memory cost, and capacity?

4.3 How does the principle of locality relate to the use of multiple memory levels?

4.4 What are the differences among direct mapping, associative mapping, and set-associative mapping?

4.5 For a direct-mapped cache, a main memory address is viewed as consisting of three fields. List and define the three fields.

4.6 For an associative cache, a main memory address is viewed as consisting of two fields. List and define the two fields.

4.7 For a set-associative cache, a main memory address is viewed as consisting of three fields. List and define the three fields.

4.8 What is the distinction between spatial locality and temporal locality?

4.9 In general, what are the strategies for exploiting spatial locality and temporal locality?

Problems

4.1 A set-associative cache consists of 64 lines, or slots, divided into four-line sets. Main memory contains 4K blocks of 128 words each. Show the format of main memory addresses.

4.2 A two-way set-associative cache has lines of 16 bytes and a total size of 8 kB. The 64-MB main memory is byte addressable. Show the format of main memory addresses.

4.3 For the hexadecimal main memory addresses 111111, 666666, BBBBBB, show the following information, in hexadecimal format:

- **a.** Tag, Line, and Word values for a direct-mapped cache, using the format of Figure 4.10
- **b.** Tag and Word values for an associative cache, using the format of Figure 4.12
- **c.** Tag, Set, and Word values for a two-way set-associative cache, using the format of Figure 4.15

4.4 List the following values:

- **a.** For the direct cache example of Figure 4.10: address length, number of addressable units, block size, number of blocks in main memory, number of lines in cache, size of tag
- **b.** For the associative cache example of Figure 4.12: address length, number of addressable units, block size, number of blocks in main memory, number of lines in cache, size of tag

c. For the two-way set-associative cache example of Figure 4.15: address length, number of addressable units, block size, number of blocks in main memory, number of lines in set, number of sets, number of lines in cache, size of tag

4.5 Consider a 32-bit microprocessor that has an on-chip 16-kB four-way set-associative cache. Assume that the cache has a line size of four 32-bit words. Draw a block diagram of this cache showing its organization and how the different address fields are used to determine a cache hit/miss. Where in the cache is the word from memory location ABCDE8F8 mapped?

4.6 Given the following specifications for an external cache memory: four-way set associative; line size of two 16-bit words; able to accommodate a total of 4K 32-bit words from main memory; used with a 16-bit processor that issues 24-bit addresses. Design the cache structure with all pertinent information and show how it interprets the processor's addresses.

4.7 The Intel 80486 has an on-chip, unified cache. It contains 8 kB and has a four-way set-associative organization and a block length of four 32-bit words. The cache is organized into 128 sets. There is a single "line valid bit" and three bits, B0, B1, and B2 (the "LRU" bits), per line. On a cache miss, the 80486 reads a 16-byte line from main memory in a bus memory read burst. Draw a simplified diagram of the cache and show how the different fields of the address are interpreted.

4.8 Consider a machine with a byte addressable main memory of 2^{16} bytes and block size of 8 bytes. Assume that a direct mapped cache consisting of 32 lines is used with this machine.

a. How is a 16-bit memory address divided into tag, line number, and byte number?

b. Into what line would bytes with each of the following addresses be stored?

0001	0001	0001	1011
1100	0011	0011	0100
1101	0000	0001	1101
1010	1010	1010	1010

c. Suppose the byte with address 0001 1010 0001 1010 is stored in the cache. What are the addresses of the other bytes stored along with it?

d. How many total bytes of memory can be stored in the cache?

e. Why is the tag also stored in the cache?

4.9 For its on-chip cache, the Intel 80486 uses a replacement algorithm referred to as **pseudo least recently used**. Associated with each of the 128 sets of four lines (labeled L0, L1, L2, L3) are three bits B0, B1, and B2. The replacement algorithm works as follows: When a line must be replaced, the cache will first determine whether the most recent use was from L0 and L1 or L2 and L3. Then the cache will determine which of the pair of blocks was least recently used and mark it for replacement. Figure 4.19 illustrates the logic.

a. Specify how the bits B0, B1, and B2 are set and then describe in words how they are used in the replacement algorithm depicted in Figure 4.19.

b. Show that the 80486 algorithm approximates a true LRU algorithm. *Hint:* Consider the case in which the most recent order of usage is L0, L2, L3, L1.

c. Demonstrate that a true LRU algorithm would require 6 bits per set.

4.10 A set-associative cache has a block size of four 16-bit words and a set size of 2. The cache can accommodate a total of 4096 words. The main memory size that is cacheable is 64K $\times$ 32 bits. Design the cache structure and show how the processor's addresses are interpreted.

4.11 Consider a memory system that uses a 32-bit address to address at the byte level, plus a cache that uses a 64-byte line size.

a. Assume a direct mapped cache with a tag field in the address of 20 bits. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in cache, size of tag.

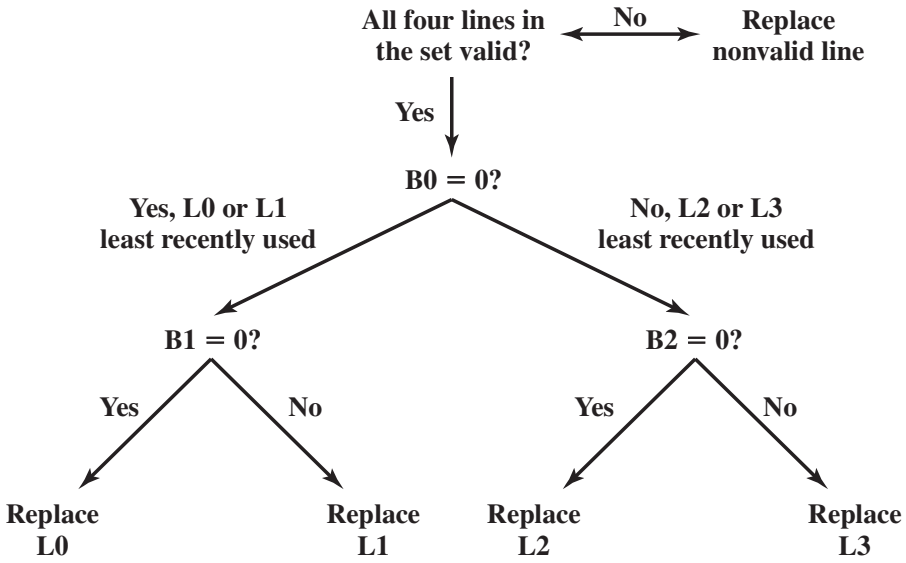


Figure 4.19 Intel 80486 On-Chip Cache Replacement Strategy

b. Assume an associative cache. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in cache, size of tag.

c. Assume a four-way set-associative cache with a tag field in the address of 9 bits. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in set, number of sets in cache, number of lines in cache, size of tag.

4.12 Consider a computer with the following characteristics: total of 1 MB of main memory; word size of 1 byte; block size of 16 bytes; and cache size of 64 kB.

a. For the main memory addresses of F0010, 01234, and CABBE, give the corresponding tag, cache line address, and word offsets for a direct-mapped cache.

b. Give any two main memory addresses with different tags that map to the same cache slot for a direct-mapped cache.

c. For the main memory addresses of F0010 and CABBE, give the corresponding tag and offset values for a fully-associative cache.

d. For the main memory addresses of F0010 and CABBE, give the corresponding tag, cache set, and offset values for a two-way set-associative cache.

4.13 Describe a simple technique for implementing an LRU replacement algorithm in a four-way set-associative cache.

4.14 Consider again Example 4.3. How does the answer change if the main memory uses a block transfer capability that has a first-word access time of 30 ns and an access time of 5 ns for each word thereafter?

4.15 Consider the following code:

```
for (i = 0; i < 20; i++)
    for (j = 0; j < 10; j++)
        a[i] = a[i]*j
```

a. Give one example of the spatial locality in the code.

b. Give one example of the temporal locality in the code.

4.16 Generalize Equations (4.2) and (4.3), in Appendix 4A, to *N*-level memory hierarchies.

4.17 A computer system contains a main memory of 32K 16-bit words. It also has a 4K word cache divided into four-line sets with 64 words per line. Assume that the cache is initially empty. The processor fetches words from locations 0, 1, 2, . . . , 4351 in that

order. It then repeats this fetch sequence nine more times. The cache is 10 times faster than main memory. Estimate the improvement resulting from the use of the cache. Assume an LRU policy for block replacement.

4.18 Consider a cache of 4 lines of 16 bytes each. Main memory is divided into blocks of 16 bytes each. That is, block 0 has bytes with addresses 0 through 15, and so on. Now consider a program that accesses memory in the following sequence of addresses:
Once: 63 through 70.
Loop ten times: 15 through 32; 80 through 95.

a. Suppose the cache is organized as direct mapped. Memory blocks 0, 4, and so on are assigned to line 1; blocks 1, 5, and so on to line 2; and so on. Compute the hit ratio.

b. Suppose the cache is organized as two-way set associative, with two sets of two lines each. Even-numbered blocks are assigned to set 0 and odd-numbered blocks are assigned to set 1. Compute the hit ratio for the two-way set-associative cache using the least recently used replacement scheme.

4.19 Consider a memory system with the following parameters:

$$T_c = 100 \text{ ns} \qquad C_c = 10^{-4} \text{ \$/bit}$$

$$T_m = 1200 \text{ ns} \qquad C_m = 10^{-5} \text{ \$/bit}$$

a. What is the cost of 1 MB of main memory?

b. What is the cost of 1 MB of main memory using cache memory technology?

c. If the effective access time is 10% greater than the cache access time, what is the hit ratio H?

4.20 **a.** Consider an L1 cache with an access time of 1 ns and a hit ratio of $H = 0.95$. Suppose that we can change the cache design (size of cache, cache organization) such that we increase H to 0.97, but increase access time to 1.5 ns. What conditions must be met for this change to result in improved performance?

d. Explain why this result makes intuitive sense.

4.21 Consider a single-level cache with an access time of 2.5 ns, a line size of 64 bytes, and a hit ratio of $H = 0.95$. Main memory uses a block transfer capability that has a first-word (4 bytes) access time of 50 ns and an access time of 5 ns for each word thereafter.

a. What is the access time when there is a cache miss? Assume that the cache waits until the line has been fetched from main memory and then re-executes for a hit.

b. Suppose that increasing the line size to 128 bytes increases the H to 0.97. Does this reduce the average memory access time?

4.22 A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20 ns are required to access it. If it is in main memory but not in the cache, 60 ns are needed to load it into the cache, and then the reference is started again. If the word is not in main memory, 12 ms are required to fetch the word from disk, followed by 60 ns to copy it to the cache, and then the reference is started again. The cache hit ratio is 0.9 and the main memory hit ratio is 0.6. What is the average time in nanoseconds required to access a referenced word on this system?

4.23 Consider a cache with a line size of 64 bytes. Assume that on average 30% of the lines in the cache are dirty. A word consists of 8 bytes.

a. Assume there is a 3% miss rate (0.97 hit ratio). Compute the amount of main memory traffic, in terms of bytes per instruction for both write-through and write-back policies. Memory is read into cache one line at a time. However, for write back, a single word can be written from cache to main memory.

b. Repeat part a for a 5% rate.

c. Repeat part a for a 7% rate.

d. What conclusion can you draw from these results?

4.24 On the Motorola 68020 microprocessor, a cache access takes two clock cycles. Data access from main memory over the bus to the processor takes three clock cycles in the

case of no wait state insertion; the data are delivered to the processor in parallel with delivery to the cache.

a. Calculate the effective length of a memory cycle given a hit ratio of 0.9 and a clocking rate of 16.67 MHz.

b. Repeat the calculations assuming insertion of two wait states of one cycle each per memory cycle. What conclusion can you draw from the results?

4.25 Assume a processor having a memory cycle time of 300 ns and an instruction processing rate of 1 MIPS. On average, each instruction requires one bus memory cycle for instruction fetch and one for the operand it involves.

a. Calculate the utilization of the bus by the processor.

b. Suppose the processor is equipped with an instruction cache and the associated hit ratio is 0.5. Determine the impact on bus utilization.

4.26 The performance of a single-level cache system for a read operation can be characterized by the following equation:

$$T_a = T_c + (1 - H)T_m$$

where T_a is the average access time, T_c is the cache access time, T_m is the memory access time (memory to processor register), and H is the hit ratio. For simplicity, we assume that the word in question is loaded into the cache in parallel with the load to processor register. This is the same form as Equation (4.2).

a. Define T_b = time to transfer a line between cache and main memory, and W = fraction of write references. Revise the preceding equation to account for writes as well as reads, using a write-through policy.

b. Define W_b as the probability that a line in the cache has been altered. Provide an equation for T_a for the write-back policy.

4.27 For a system with two levels of cache, define T_{c_1} = first − level cache access time; T_{c_2} = second − level cache access time; T_m = memory access time; H_1 = first − level cache hit ratio; H_2 = combined first/second level cache hit ratio. Provide an equation for T_a for a read operation.

4.28 Assume the following performance characteristics on a cache read miss: one clock cycle to send an address to main memory and four clock cycles to access a 32-bit word from main memory and transfer it to the processor and cache.

a. If the cache line size is one word, what is the miss penalty (i.e., additional time required for a read in the event of a read miss)?

b. What is the miss penalty if the cache line size is four words and a multiple, nonburst transfer is executed?

c. What is the miss penalty if the cache line size is four words and a transfer is executed, with one clock cycle per word transfer?

4.29 For the cache design of the preceding problem, suppose that increasing the line size from one word to four words results in a decrease of the read miss rate from 3.2% to 1.1%. For both the nonburst transfer and the burst transfer case, what is the average miss penalty, averaged over all reads, for the two different line sizes?

APPENDIX 4A PERFORMANCE CHARACTERISTICS OF TWO-LEVEL MEMORIES

In this chapter, reference is made to a cache that acts as a buffer between main memory and processor, creating a two-level internal memory. This two-level architecture exploits a property known as locality to provide improved performance over a comparable one-level memory.

The main memory cache mechanism is part of the computer architecture, implemented in hardware and typically invisible to the operating system. There are two other instances of a two-level memory approach that also exploit locality and that are, at least partially, implemented in the operating system: virtual memory and the disk cache (Table 4.6). Virtual memory is explored in Chapter 8; disk cache is beyond the scope of this book but is examined in [STAL15]. In this appendix, we look at some of the performance characteristics of two-level memories that are common to all three approaches.

Locality

The basis for the performance advantage of a two-level memory is a principle known as **locality of reference** [DENN68]. This principle states that memory references tend to cluster. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

Intuitively, the principle of locality makes sense. Consider the following line of reasoning:

1. Except for branch and call instructions, which constitute only a small fraction of all program instructions, program execution is sequential. Hence, in most cases, the next instruction to be fetched immediately follows the last instruction fetched.
2. It is rare to have a long uninterrupted sequence of procedure calls followed by the corresponding sequence of returns. Rather, a program remains confined to a rather narrow window of procedure-invocation depth. Thus, over a short period of time references to instructions tend to be localized to a few procedures.
3. Most iterative constructs consist of a relatively small number of instructions repeated many times. For the duration of the iteration, computation is therefore confined to a small contiguous portion of a program.
4. In many programs, much of the computation involves processing data structures, such as arrays or sequences of records. In many cases, successive references to these data structures will be to closely located data items.

Table 4.6 Characteristics of Two-Level Memories

	Main Memory Cache	Virtual Memory (paging)	Disk Cache
Typical access time ratios	5:1 (main memory vs. cache)	10^6:1 (main memory vs. disk)	10^6:1 (main memory vs. disk)
Memory management system	Implemented by special hardware	Combination of hardware and system software	System software
Typical block or page size	4 to 128 bytes (cache block)	64 to 4096 bytes (virtual memory page)	64 to 4096 bytes (disk block or pages)
Access of processor to second level	Direct access	Indirect access	Indirect access

Table 4.7 Relative Dynamic Frequency of High-Level Language Operations

Study	[HUCK83]	[KNUT71]	[PATT82a]		[TANE78]
Language	Pascal	FORTRAN	Pascal	C	SAL
Workload	Scientific	Student	System	System	System
Assign	74	67	45	38	42
Loop	4	3	5	3	4
Call	1	3	15	12	12
IF	20	11	29	43	36
GOTO	2	9	—	3	—
Other	—	7	6	1	6

This line of reasoning has been confirmed in many studies. With reference to point 1, a variety of studies have analyzed the behavior of high-level language programs. Table 4.7 includes key results, measuring the appearance of various statement types during execution, from the following studies. The earliest study of programming language behavior, performed by Knuth [KNUT71], examined a collection of FORTRAN programs used as student exercises. Tanenbaum [TANE78] published measurements collected from over 300 procedures used in operating-system programs and written in a language that supports structured programming (SAL). Patterson and Sequein [PATT82a] analyzed a set of measurements taken from compilers and programs for typesetting, computer-aided design (CAD), sorting, and file comparison. The programming languages C and Pascal were studied. Huck [HUCK83] analyzed four programs intended to represent a mix of general-purpose scientific computing, including fast Fourier transform and the integration of systems of differential equations. There is good agreement in the results of this mixture of languages and applications that branching and call instructions represent only a fraction of statements executed during the lifetime of a program. Thus, these studies confirm assertion 1.

With respect to assertion 2, studies reported in [PATT85a] provide confirmation. This is illustrated in Figure 4.20, which shows call-return behavior. Each call is represented by the line moving down and to the right, and each return by the line moving up and to the right. In the figure, a *window* with depth equal to 5 is defined. Only a sequence of calls and returns with a net movement of 6 in either direction causes the window to move. As can be seen, the executing program can remain within a stationary window for long periods of time. A study by the same analysts of C and Pascal programs showed that a window of depth 8 will need to shift only on less than 1% of the calls or returns [TAMI83].

A distinction is made in the literature between spatial locality and temporal locality. **Spatial locality** refers to the tendency of execution to involve a number of memory locations that are clustered. This reflects the tendency of a processor to access instructions sequentially. Spatial location also reflects the tendency of a program to access data locations sequentially, such as when processing a table of data. **Temporal locality** refers to the tendency for a processor to access memory locations that have been used recently. For example, when an iteration loop is executed, the processor executes the same set of instructions repeatedly.

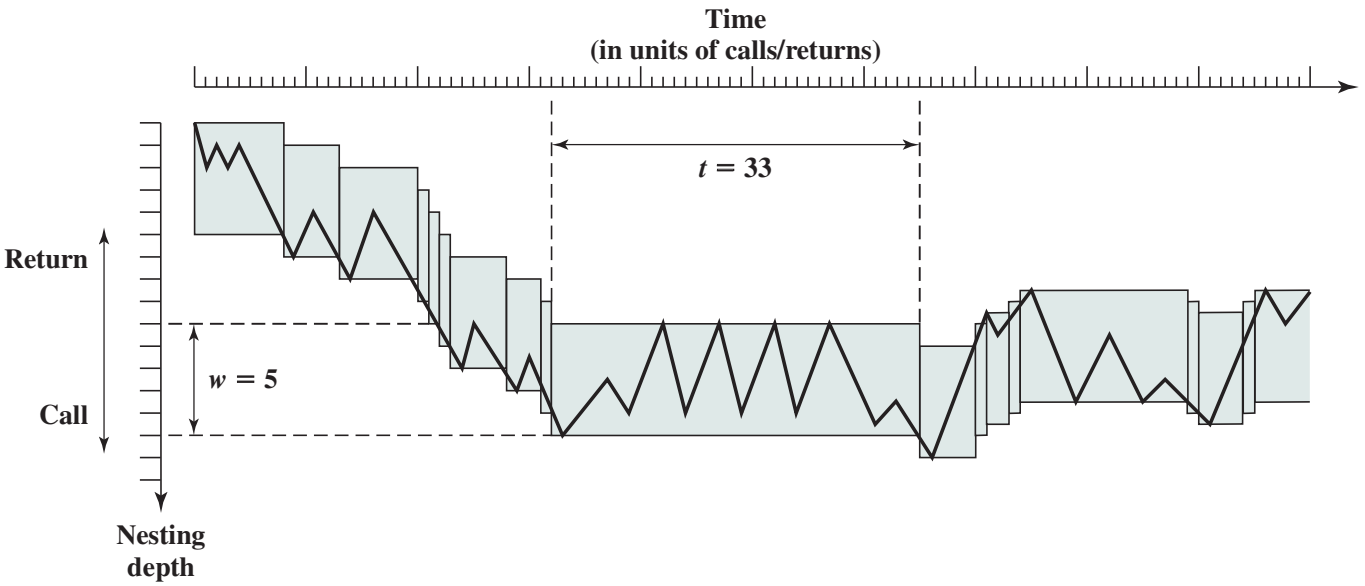


Figure 4.20 Example Call-Return Behavior of a Program

Traditionally, temporal locality is exploited by keeping recently used instruction and data values in cache memory and by exploiting a cache hierarchy. Spatial locality is generally exploited by using larger cache blocks and by incorporating prefetching mechanisms (fetching items of anticipated use) into the cache control logic. Recently, there has been considerable research on refining these techniques to achieve greater performance, but the basic strategies remain the same.

Operation of Two-Level Memory

The locality property can be exploited in the formation of a two-level memory. The upper-level memory (M1) is smaller, faster, and more expensive (per bit) than the lower-level memory (M2). M1 is used as a temporary store for part of the contents of the larger M2. When a memory reference is made, an attempt is made to access the item in M1. If this succeeds, then a quick access is made. If not, then a block of memory locations is copied from M2 to M1 and the access then takes place via M1. Because of locality, once a block is brought into M1, there should be a number of accesses to locations in that block, resulting in fast overall service.

To express the average time to access an item, we must consider not only the speeds of the two levels of memory, but also the probability that a given reference can be found in M1. We have

$$\begin{aligned} T_s &= H \times T_1 + (1 - H) \times (T_1 + T_2) \\ &= T_1 + (1 - H) \times T_2 \end{aligned} \tag{4.2}$$

where

T_s = average (system) access time
T_1 = access time of M1 (e.g., *cache*, *disk cache*)
T_2 = access time of M2 (e.g., *main memory*, *disk*)
H = hit ratio (fraction of time reference is found in M1)

Figure 4.2 shows average access time as a function of hit ratio. As can be seen, for a high percentage of hits, the average total access time is much closer to that of M1 than M2.

Performance

Let us look at some of the parameters relevant to an assessment of a two-level memory mechanism. First consider cost. We have

$$C_s = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2} \tag{4.3}$$

where

C_s = average cost per bit for the combined two-level memory
C_1 = average cost per bit of upper-level memory M1
C_2 = average cost per bit of lower-level memory M2
S_1 = size of M1
S_2 = size of M2

We would like $C_s \approx C_2$. Given that $C_1 \gg C_2$, this requires $S_1 < S_2$. Figure 4.21 shows the relationship.

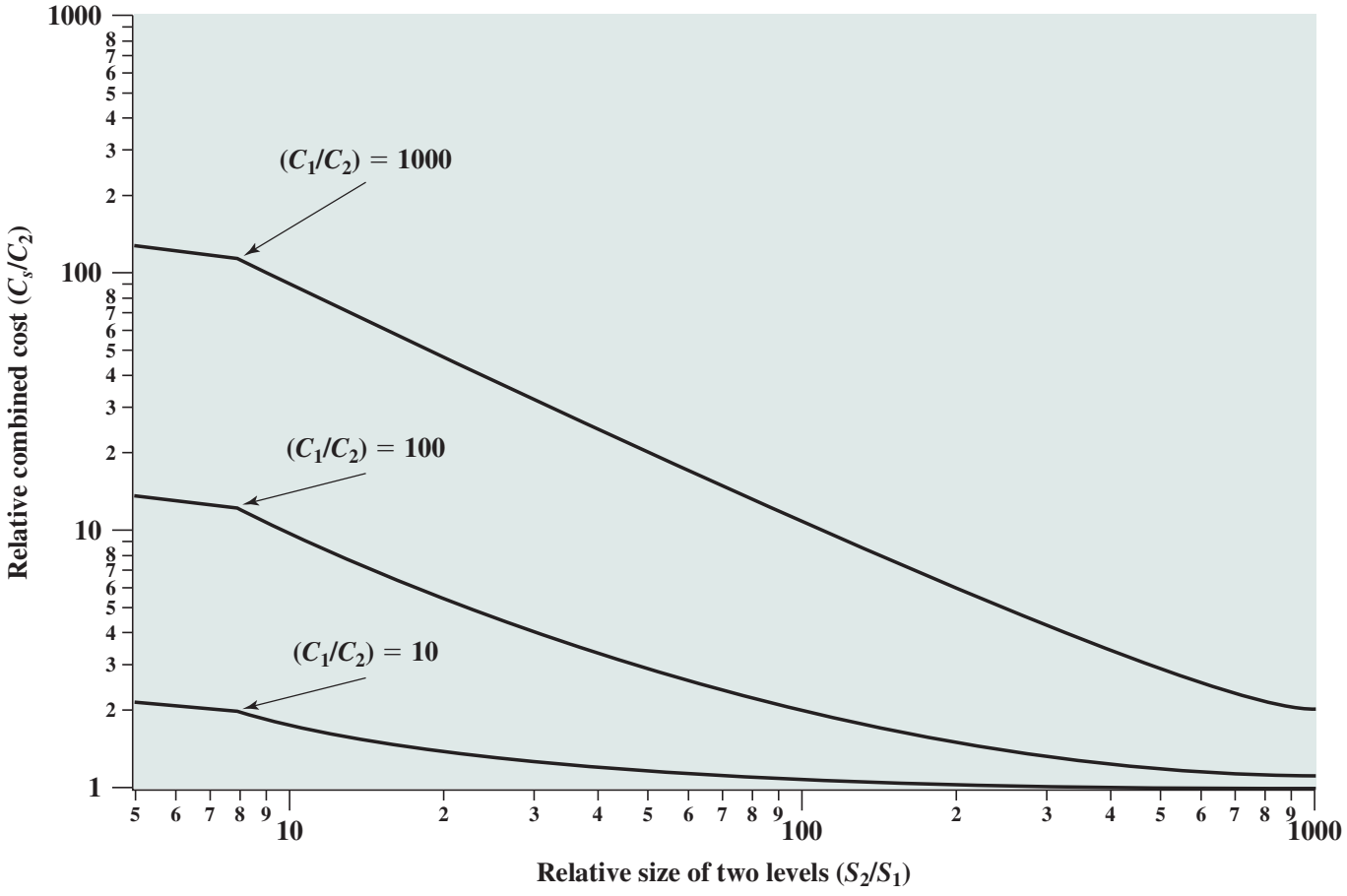


Figure 4.21 Relationship of Average Memory Cost to Relative Memory Size for a Two-Level Memory

Next, consider access time. For a two-level memory to provide a significant performance improvement, we need to have T_s approximately equal to T_1 ($T_s \approx T_1$). Given that T_1 is much less than T_2 ($T_1 << T_2$), a hit ratio of close to 1 is needed.

So we would like M1 to be small to hold down cost, and large to improve the hit ratio and therefore the performance. Is there a size of M1 that satisfies both requirements to a reasonable extent? We can answer this question with a series of subquestions:

- What value of hit ratio is needed so that $T_s \approx T_1$?
- What size of M1 will assure the needed hit ratio?
- Does this size satisfy the cost requirement?

To get at this, consider the quantity T_1/T_s, which is referred to as the *access efficiency*. It is a measure of how close average access time (T_s) is to M1 access time (T_1). From Equation (4.2),

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H)\dfrac{T_2}{T_1}} \tag{4.4}$$

Figure 4.22 plots T_1/T_s as a function of the hit ratio H, with the quantity T_2/T_1 as a parameter. Typically, on-chip cache access time is about 25 to 50 times faster than main memory access time (i.e., T_2/T_1 is 25 to 50), off-chip cache access time

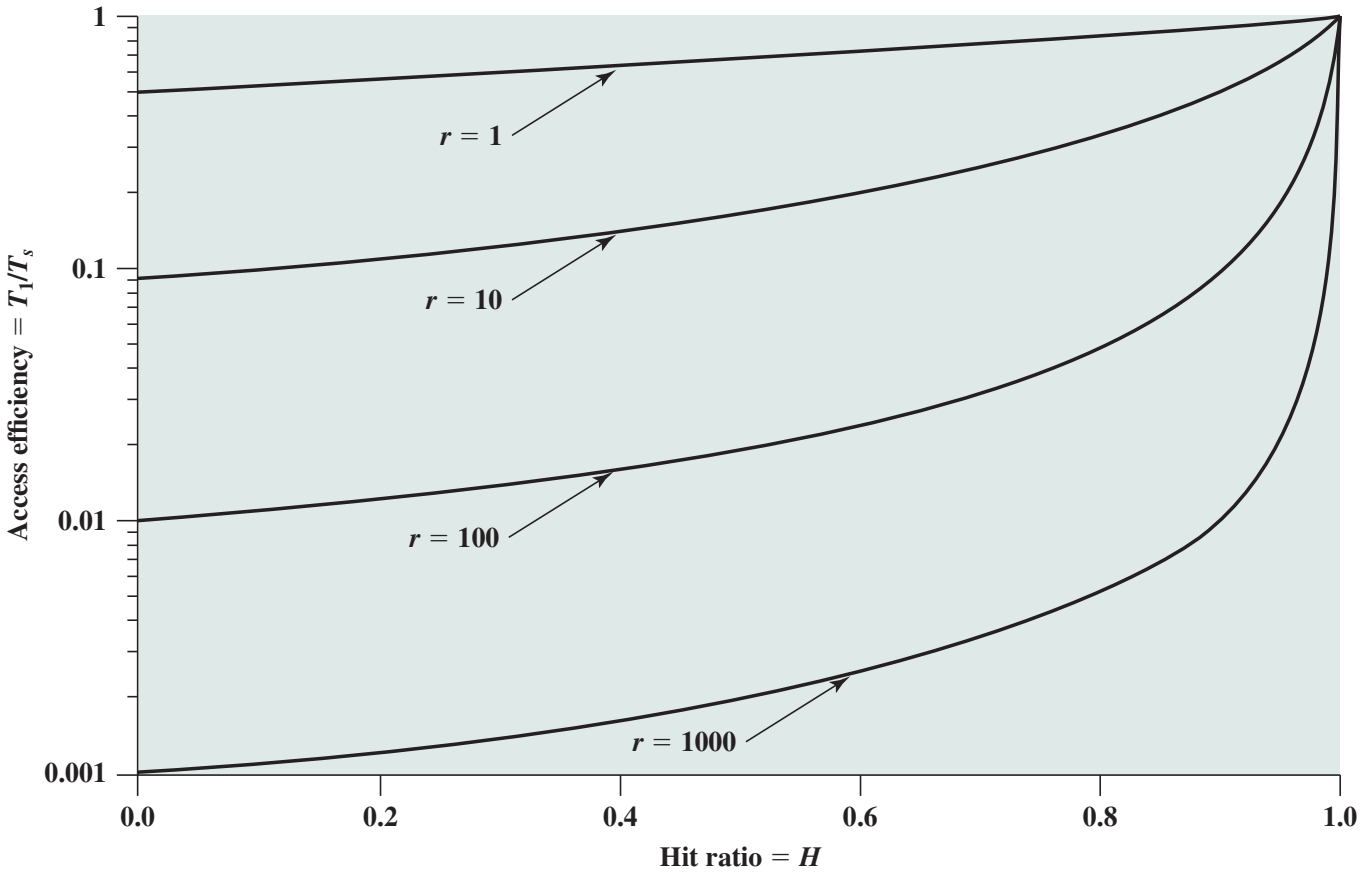


Figure 4.22 Access Efficiency as a Function of Hit Ratio ($r = T_2/T_1$)

is about 5 to 15 times faster than main memory access time (i.e., T_2/T_1 is 5 to 15), and main memory access time is about 1000 times faster than disk access time ($T_2/T_1 = 1000$). Thus, a hit ratio in the range of near 0.9 would seem to be needed to satisfy the performance requirement.

We can now phrase the question about relative memory size more exactly. Is a hit ratio of, say, 0.8 or better reasonable for $S_1 << S_2$? This will depend on a number of factors, including the nature of the software being executed and the details of the design of the two-level memory. The main determinant is, of course, the degree of locality. Figure 4.23 suggests the effect that locality has on the hit ratio. Clearly, if M1 is the same size as M2, then the hit ratio will be 1.0: All of the items in M2 are always also stored in M1. Now suppose that there is no locality; that is, references are completely random. In that case the hit ratio should be a strictly linear function of the relative memory size. For example, if M1 is half the size of M2, then at any time half of the items from M2 are also in M1 and the hit ratio will be 0.5. In practice, however, there is some degree of locality in the references. The effects of moderate and strong locality are indicated in the figure. Note that Figure 4.23 is not derived from any specific data or model; the figure suggests the type of performance that is seen with various degrees of locality.

So if there is strong locality, it is possible to achieve high values of hit ratio even with relatively small upper-level memory size. For example, numerous studies have shown that rather small cache sizes will yield a hit ratio above 0.75 *regardless of the size of main memory* (e.g., [AGAR89], [PRZY88], [STRE83], and [SMIT82]). A cache in the range of 1K to 128K words is generally adequate, whereas main

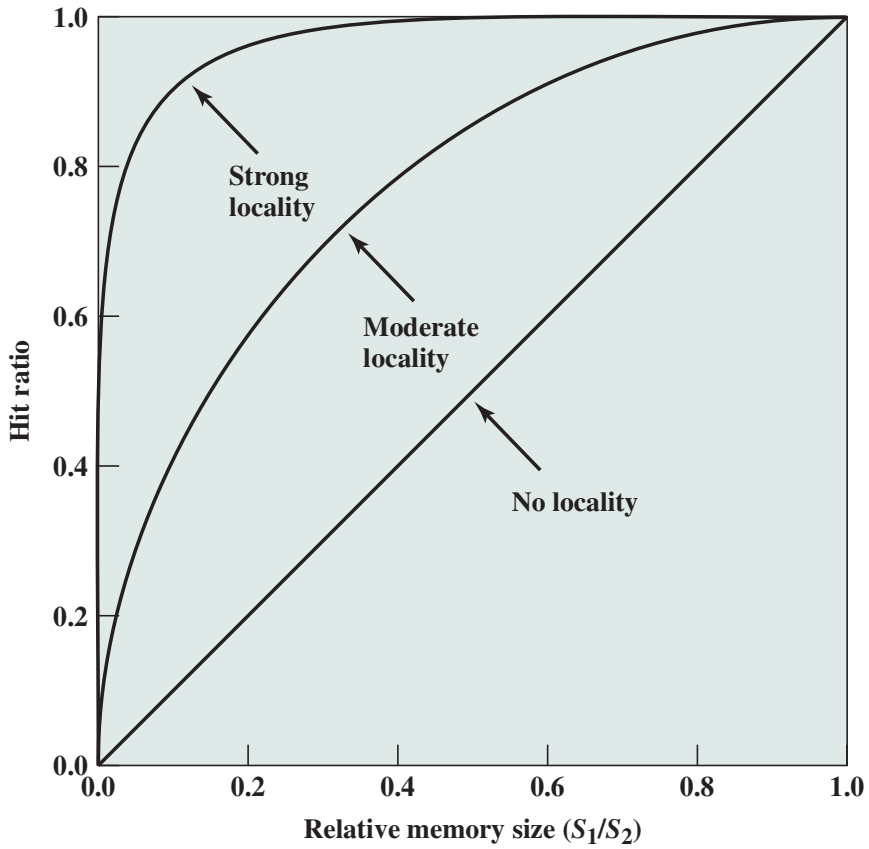


Figure 4.23 Hit Ratio as a Function of Relative Memory Size

memory is now typically in the gigabyte range. When we consider virtual memory and disk cache, we will cite other studies that confirm the same phenomenon, namely that a relatively small M1 yields a high value of hit ratio because of locality.

This brings us to the last question listed earlier: Does the relative size of the two memories satisfy the cost requirement? The answer is clearly yes. If we need only a relatively small upper-level memory to achieve good performance, then the average cost per bit of the two levels of memory will approach that of the cheaper lower-level memory.

Please note that with L2 cache, or even L2 and L3 caches, involved, analysis is much more complex. See [PEIR99] and [HAND98] for discussions.