

Chapter 5

Large and Fast: Exploiting Memory Hierarchy

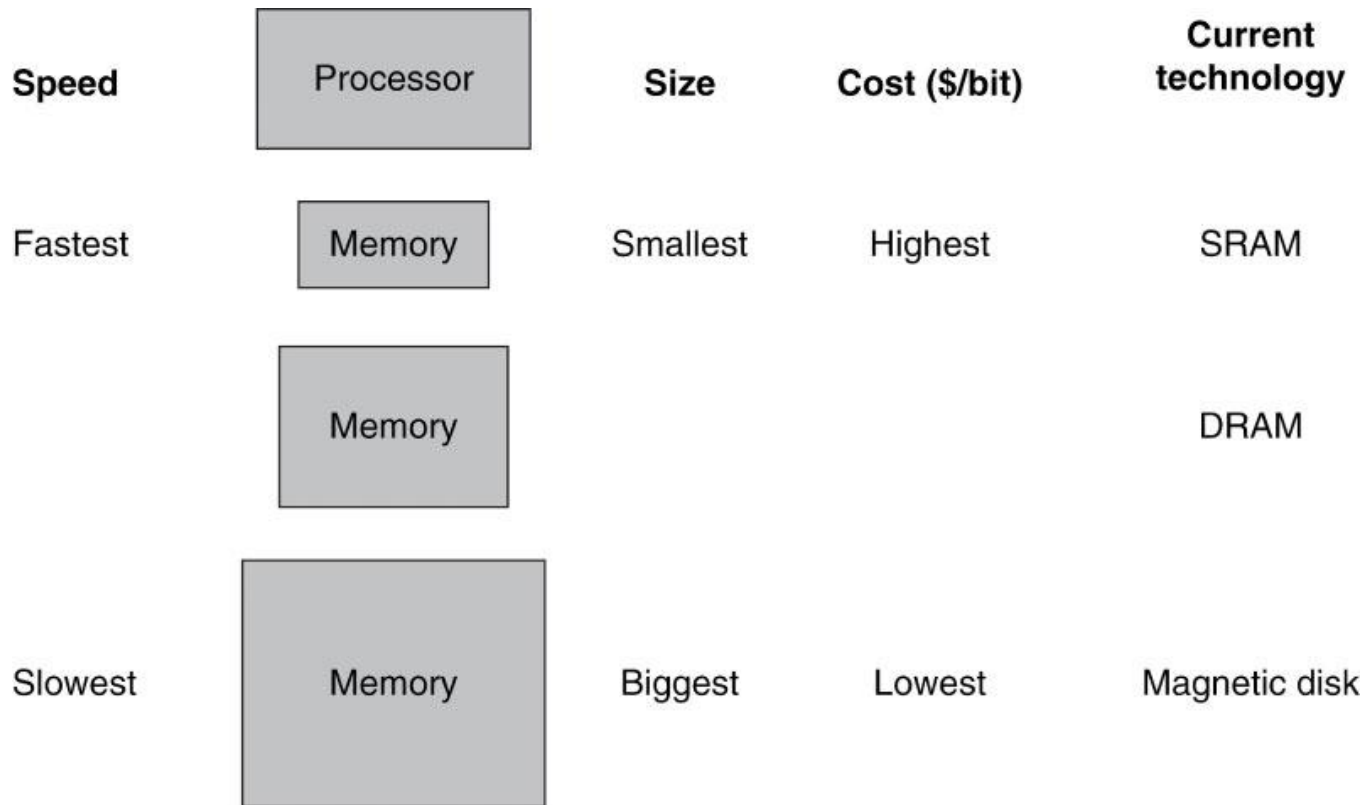


FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in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

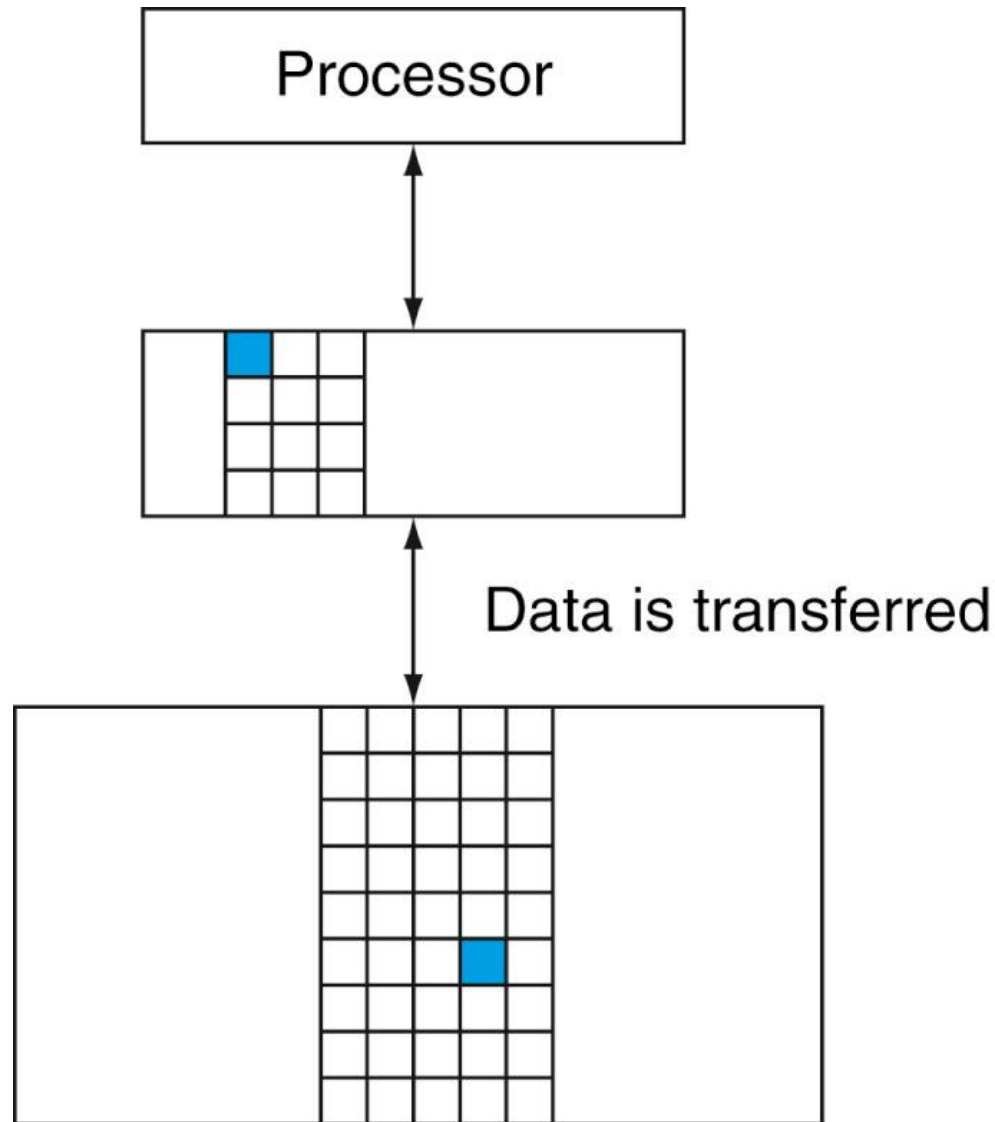


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a *block* or a *line*. Usually we transfer an entire block when we copy something between levels.

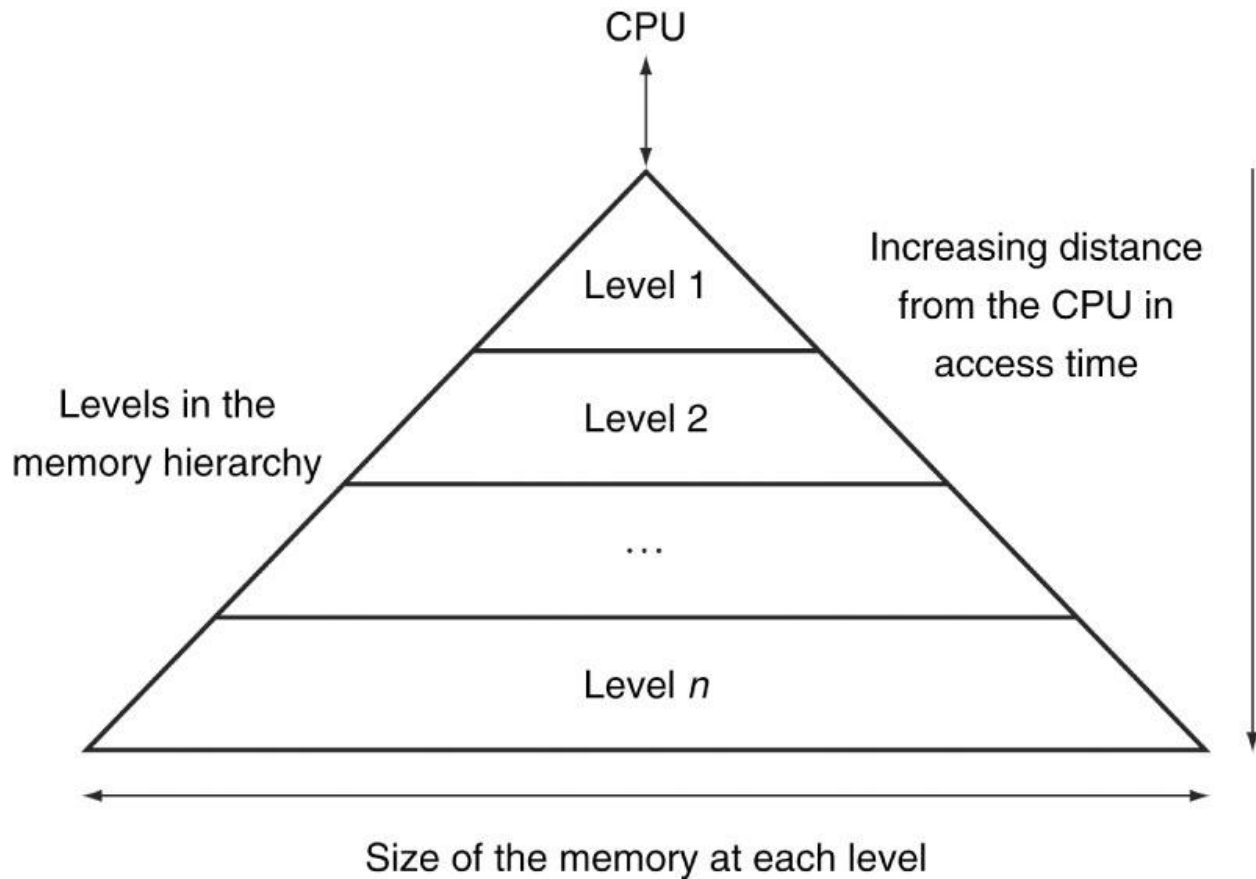


FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size. This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have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

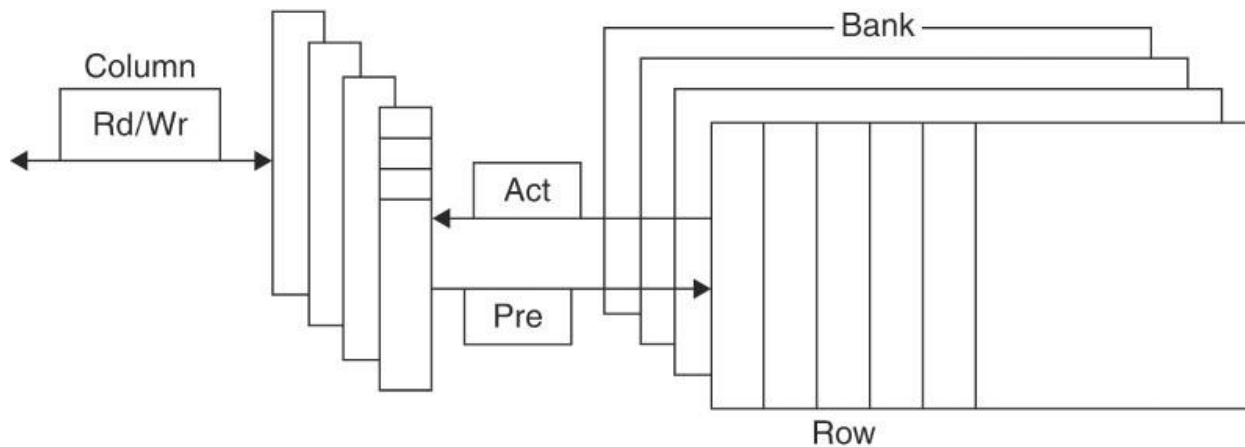


FIGURE 5.4 Internal organization of a DRAM. Modern DRAMs are organized in banks, typically four for DDR3. Each bank consists of a series of rows. Sending a PRE (precharge) command opens or closes a bank. A row address is sent with an Act (activate), which causes the row to transfer to a buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR3) or by specifying a block transfer and the starting address. Each command, as well as block transfers, is synchronized with a clock.

Year introduced	Chip size	\$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	\$1,500,000	250 ns	150 ns
1983	256 Kibibit	\$500,000	185 ns	100 ns
1985	1 Mebibit	\$200,000	135 ns	40 ns
1989	4 Mebibit	\$50,000	110 ns	40 ns
1992	16 Mebibit	\$15,000	90 ns	30 ns
1996	64 Mebibit	\$10,000	60 ns	12 ns
1998	128 Mebibit	\$4,000	60 ns	10 ns
2000	256 Mebibit	\$1,000	55 ns	7 ns
2004	512 Mebibit	\$250	50 ns	5 ns
2007	1 Gibibit	\$50	45 ns	1.25 ns
2010	2 Gibibit	\$30	40 ns	1 ns
2012	4 Gibibit	\$1	35 ns	0.8 ns

FIGURE 5.5 DRAM size increased by multiples of four approximately once every three years until 1996, and thereafter considerably slower. The improvements in access time have been slower but continuous, and cost roughly tracks density improvements, although cost is often affected by other issues, such as availability and demand. The cost per gibibyte is not adjusted for inflation.

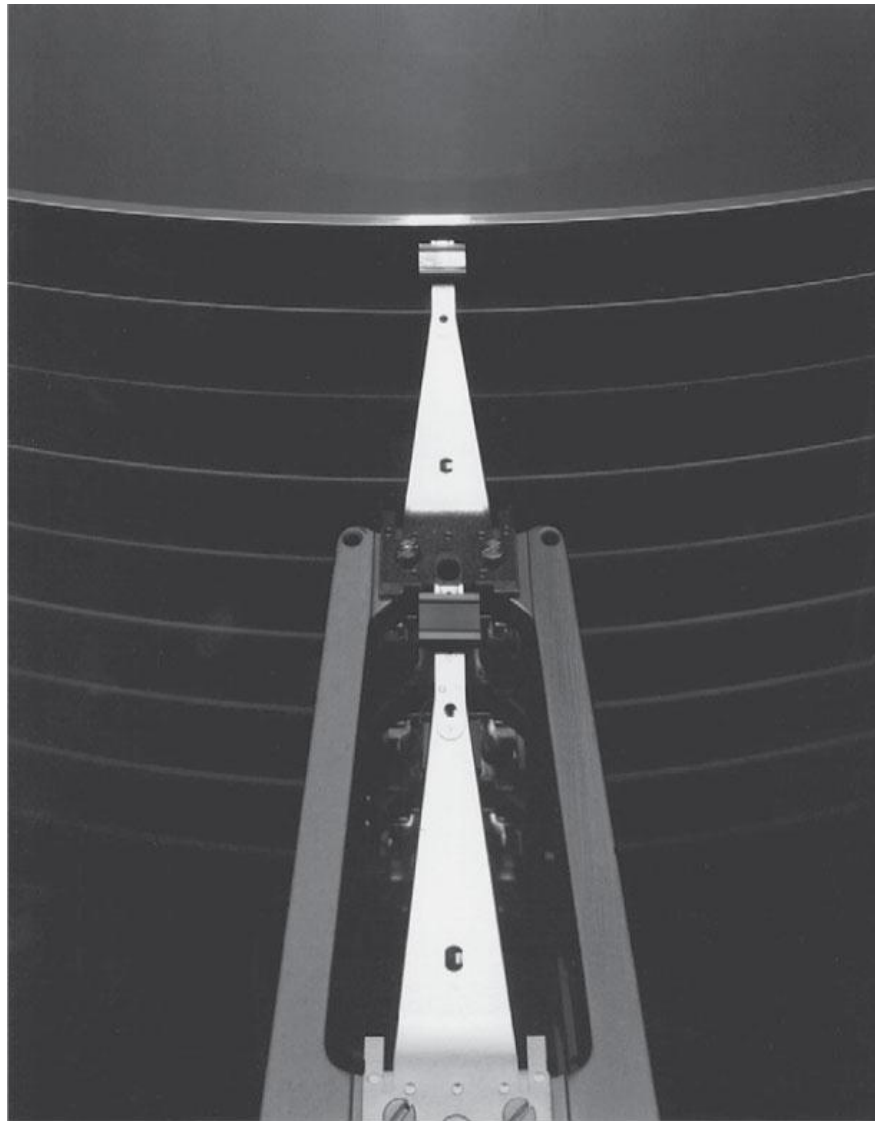
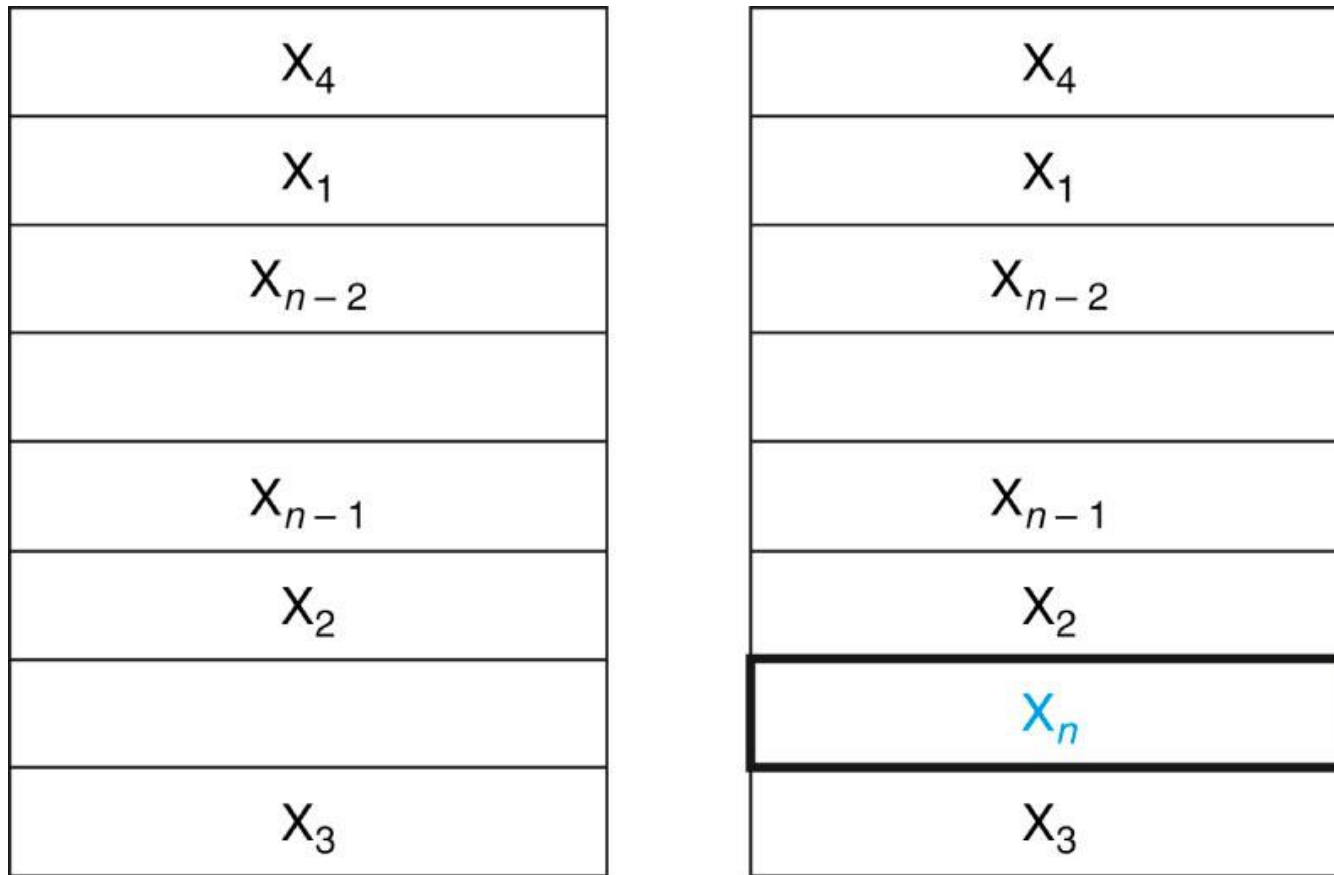


FIGURE 5.6 A disk showing 10 disk platters and the read/write heads. The diameter of today's disks is 2.5 or 3.5 inches, and there are typically one or two platters per drive today.



a. Before the reference to X_n

b. After the reference to X_n

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

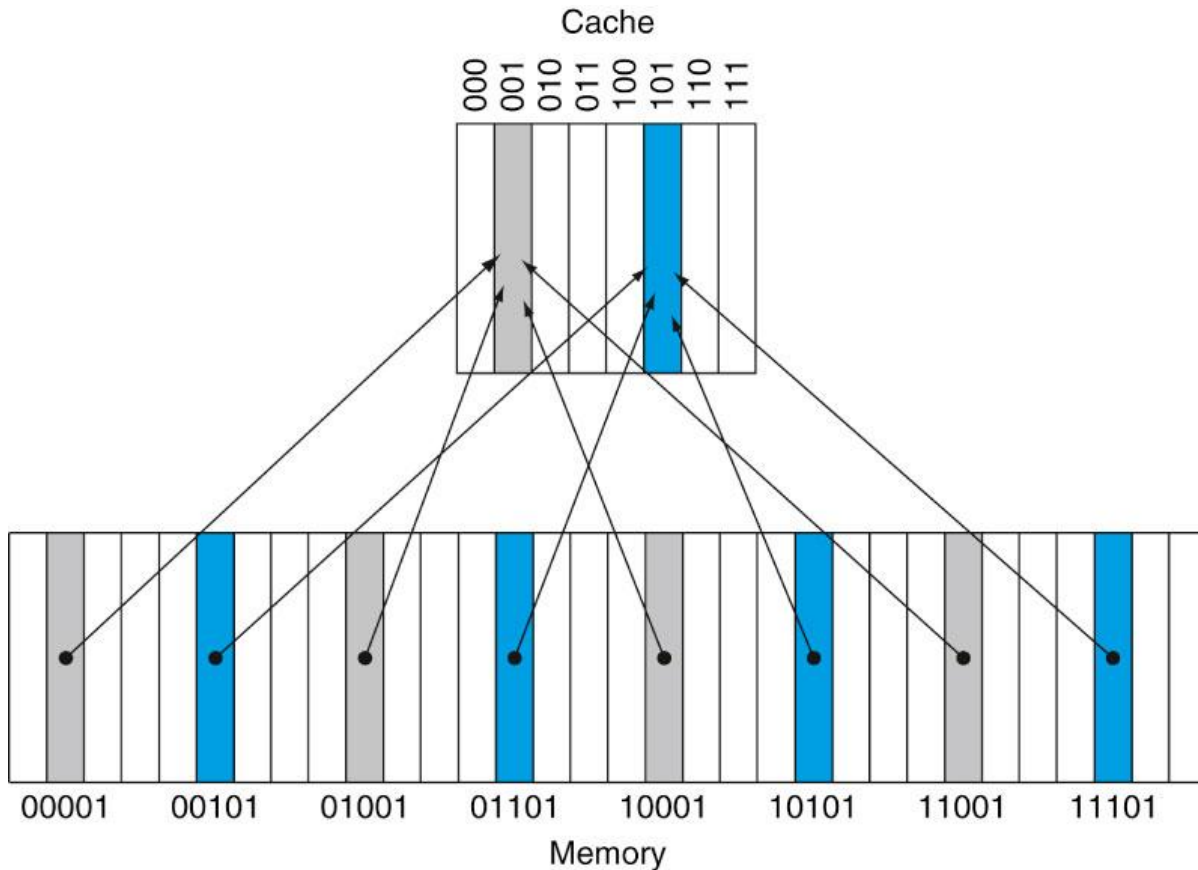


FIGURE 5.8 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the direct-mapped cache word $X \bmod 8$. That is, the low-order $\log_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001_{two} , 01001_{two} , 10001_{two} , and 11001_{two} all map to entry 001_{two} of the cache, while addresses 00101_{two} , 01101_{two} , 10101_{two} , and 11101_{two} all map to entry 101_{two} of the cache.

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

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

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

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

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

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

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

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

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

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

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

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

FIGURE 5.9 The cache contents are shown after each reference request that misses, with the index and tag fields shown in binary for the sequence of addresses on page 386. The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), 10010_{two} (miss), and 10000_{two} (hit). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss. The tag field will contain only the upper portion of the address. The full address of a word contained in cache block i with tag field j for this cache is $j381i$, or equivalently the concatenation of the tag field j and the index i . For example, in cache f above, index 010_{two} has tag 10_{two} and corresponds to address 10010_{two}.

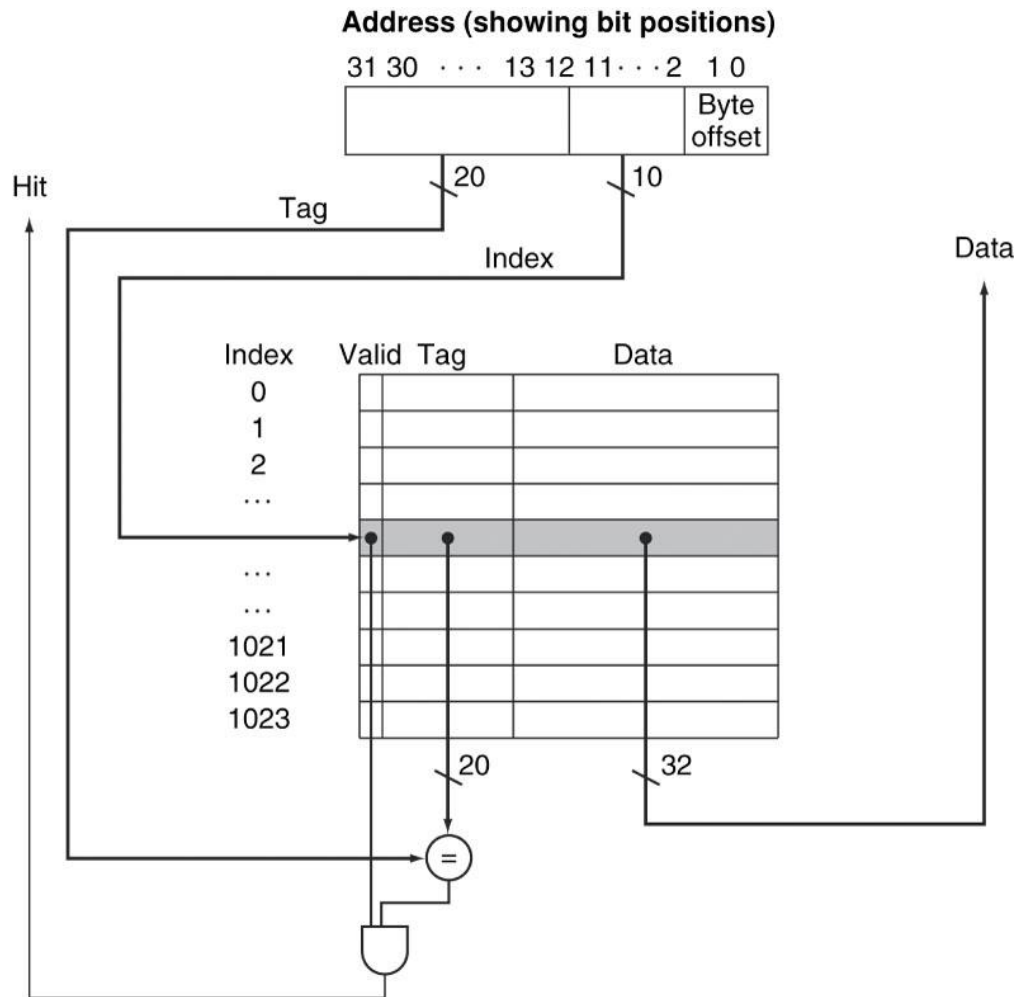


FIGURE 5.10 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. This cache holds 1024 words or 4 KiB. We assume 32-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine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

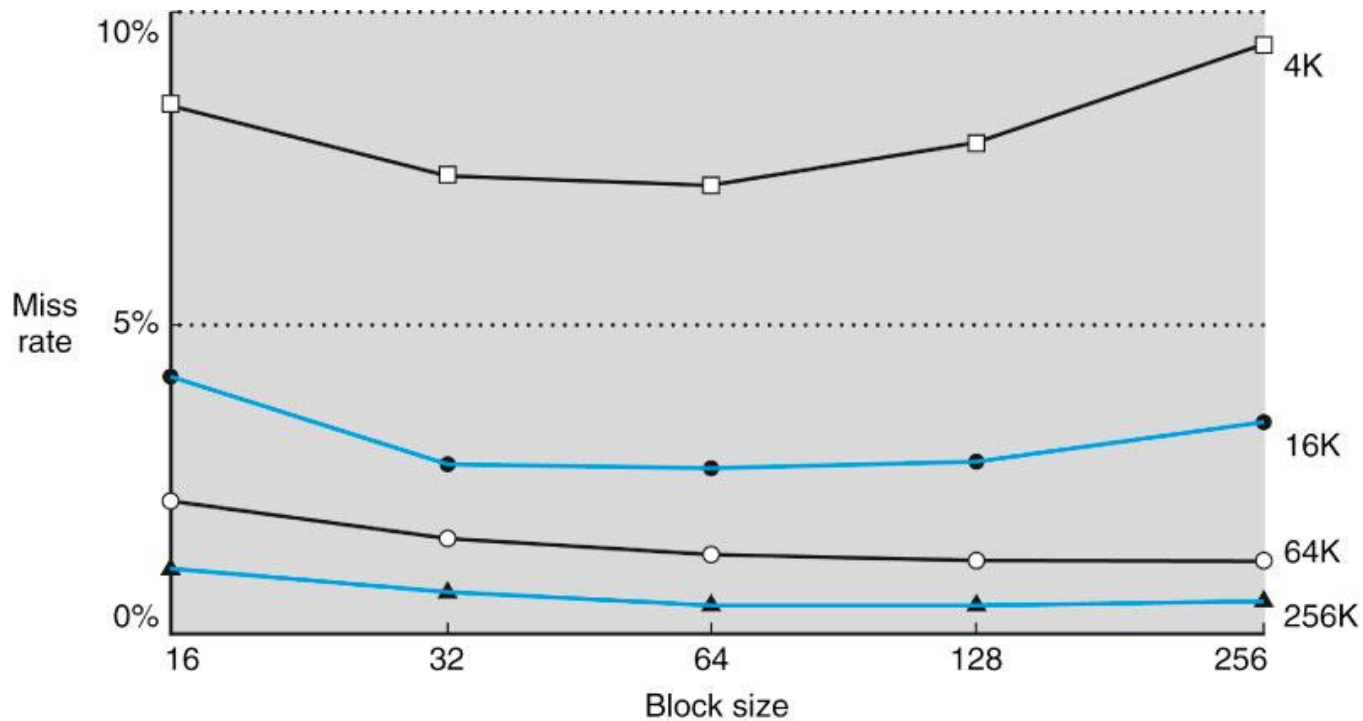


FIGURE 5.11 Miss rate versus block size. Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.) Unfortunately, SPEC CPU2000 traces would take too long if block size were included, so this data is based on SPEC92.

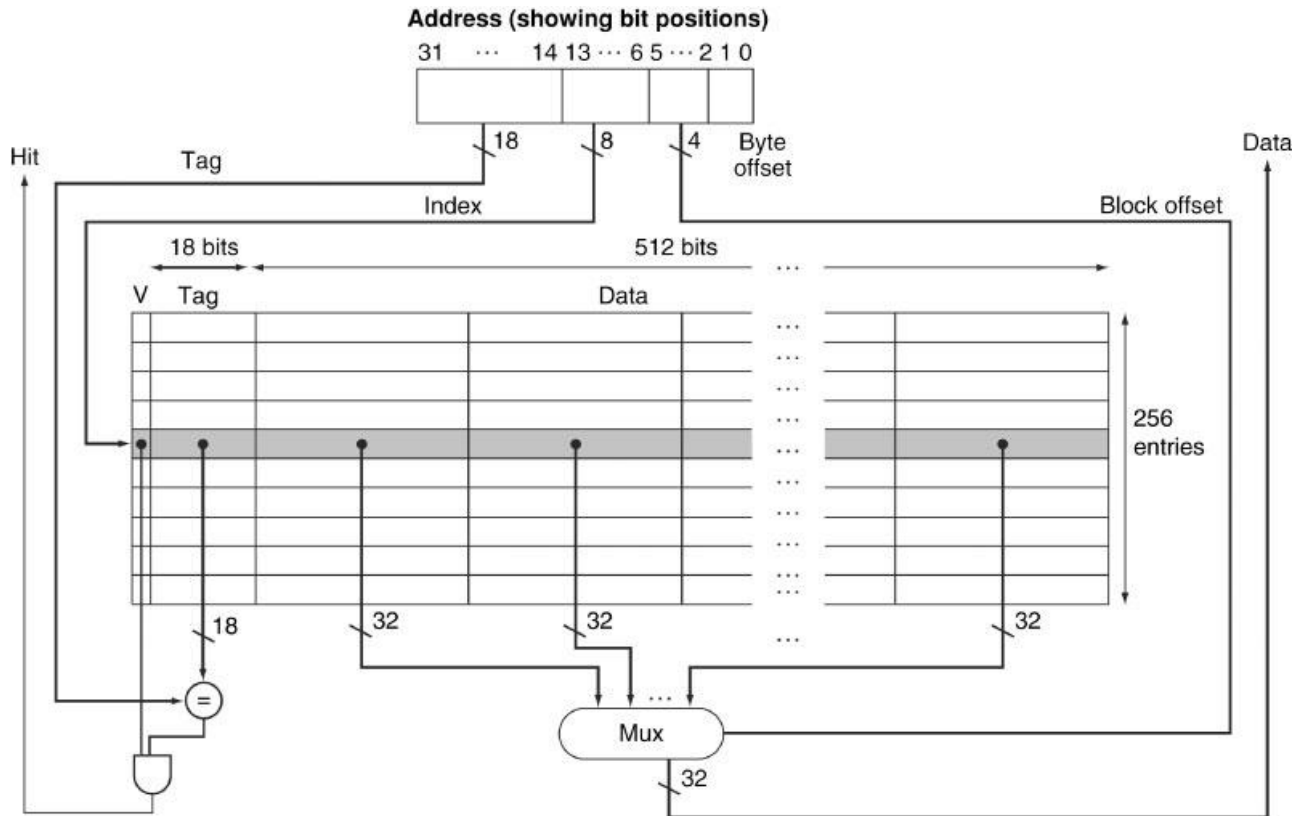


FIGURE 5.12 The 16 KiB caches in the Intrinsity FastMATH each contain 256 blocks with 16 words per block. The tag field is 18 bits wide and the index field is 8 bits wide, while a 4-bit field (bits 5–2) is used to index the block and select the word from the block using a 16-to-1 multiplexor. In practice,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

Instruction miss rate	Data miss rate	Effective combined miss rate
0.4%	11.4%	3.2%

FIGURE 5.13 Approximate instruction and data miss rates for the Intrinsity FastMATH processor for SPEC CPU2000 benchmarks. The combined miss rate is the effective miss rate seen for the combination of the 16 KiB instruction cache and 16 KiB data cache. It is obtained by weighting the instruction and data individual miss rates by the frequency of instruction and data references.

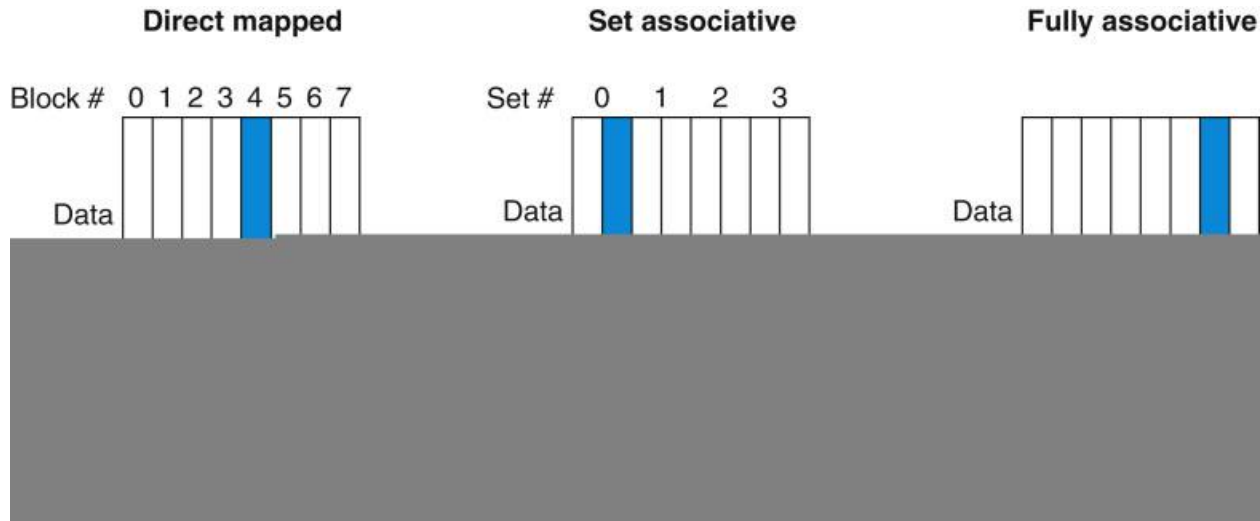


FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by $(12 \text{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

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



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

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

FIGURE 5.16 The data cache miss rates for an organization like the Intrinsity FastMATH processor for SPEC CPU2000 benchmarks with associativity varying from one-way to eight-way. These results for 10 SPEC CPU2000 programs are from Hennessy and Patterson (2003).



FIGURE 5.17 The three portions of an address in a set-associative or direct-mapped cache. The index is used to select the set, then the tag is used to choose the block by comparison with the blocks in the selected set. The block offset is the address of the desired data within the block.

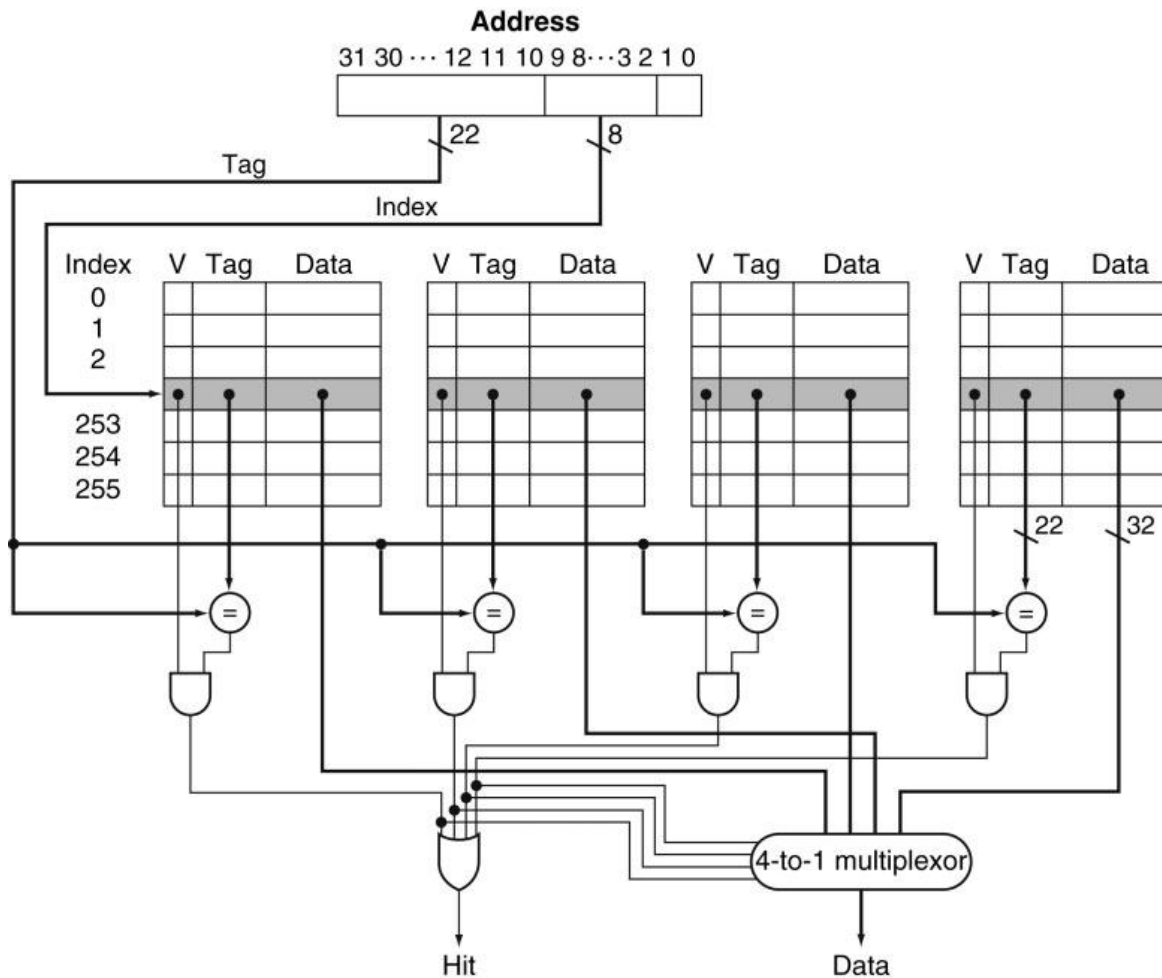


FIGURE 5.18 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor. The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed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

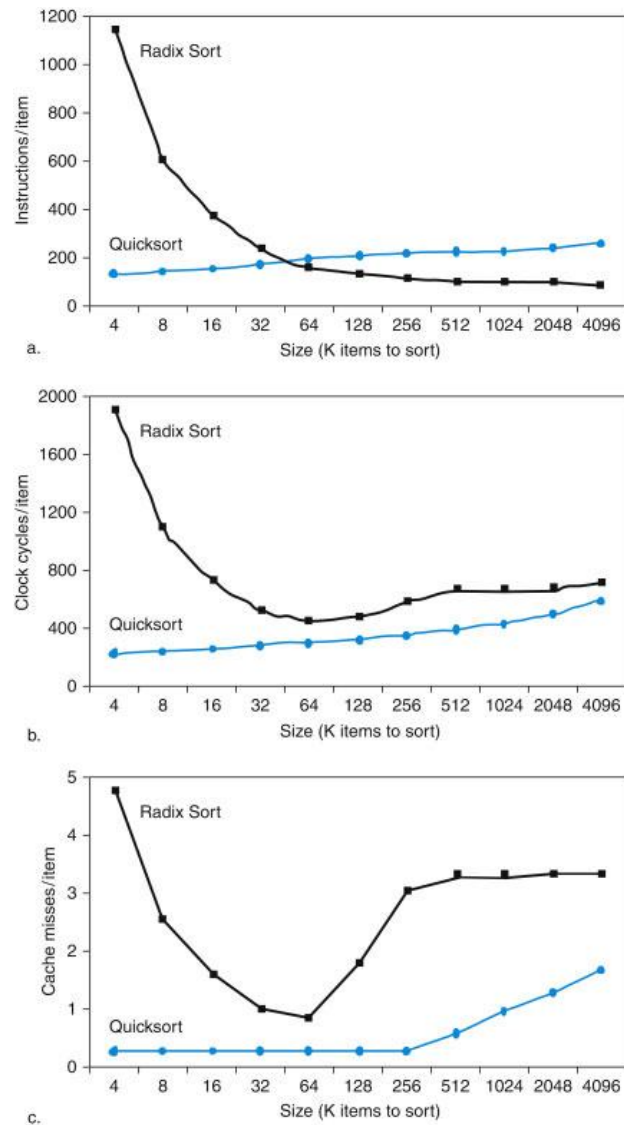


FIGURE 5.19 Comparing Quicksort and Radix Sort by (a) instructions executed per item sorted, (b) time per item sorted, and (c) cache misses per item sorted. This data is from a paper by LaMarca and Ladner [1996]. Due to such results, new versions of Radix Sort have been invented that take memory hierarchy into account, to regain its algorithmic advantages (see Section 5.15). The basic idea of cache optimizations is to use all the data in a block repeatedly before it is replaced on a miss.

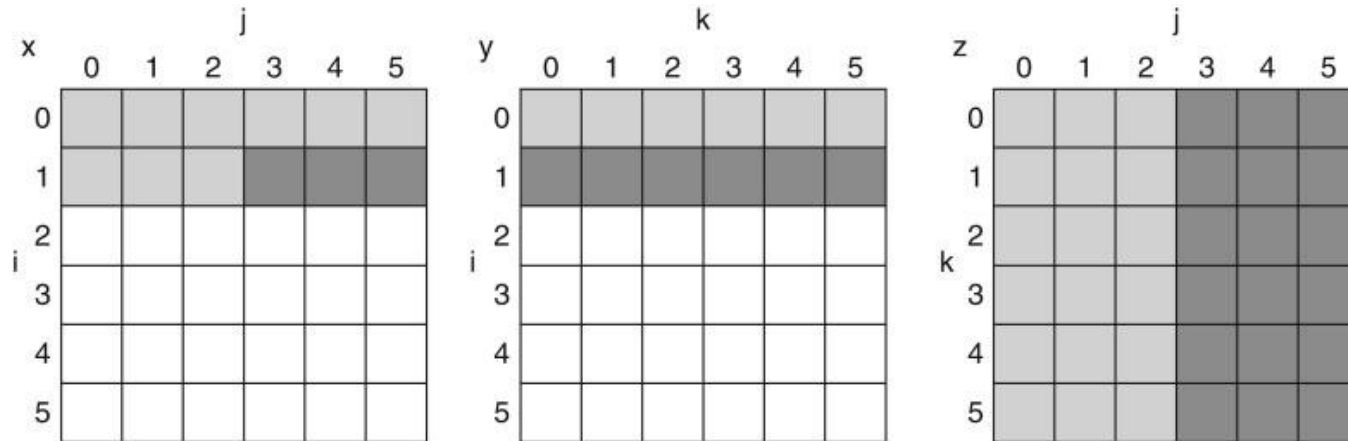


FIGURE 5.20 A snapshot of the three arrays C, A, and B when $N = 6$ and $i = 1$. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to Figure 5.21, elements of A and B are read repeatedly to calculate new elements of x. The variables i, j, and k are shown along the rows or columns used to access the arrays.

```

1  #define BLOCKSIZE 32
2  void do_block (int n, int si, int sj, int sk, double *A, double
3  *B, double *C)
4  {
5      for (int i = si; i < si+BLOCKSIZE; ++i)
6          for (int j = sj; j < sj+BLOCKSIZE; ++j)
7              {
8                  double cij = C[i+j*n]; /* cij = C[i][j] */
9                  for( int k = sk; k < sk+BLOCKSIZE; k++ )
10                     cij += A[i+k*n] * B[k+j*n]; /* cij+=A[i][k]*B[k][j] */
11                     C[i+j*n] = cij; /* C[i][j] = cij */
12             }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16     for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17         for ( int si = 0; si < n; si += BLOCKSIZE )
18             for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19                 do_block(n, si, sj, sk, A, B, C);
20 }

```

FIGURE 5.21 Cache blocked version of DGEMM in Figure 3.21. Assume C is initialized to zero. The do_block function is basically DGEMM from Chapter 3 with new parameters to specify the starting positions of the submatrices of BLOCKSIZE. The gcc optimizer can remove the function overhead instructions by inlining the do_block function.

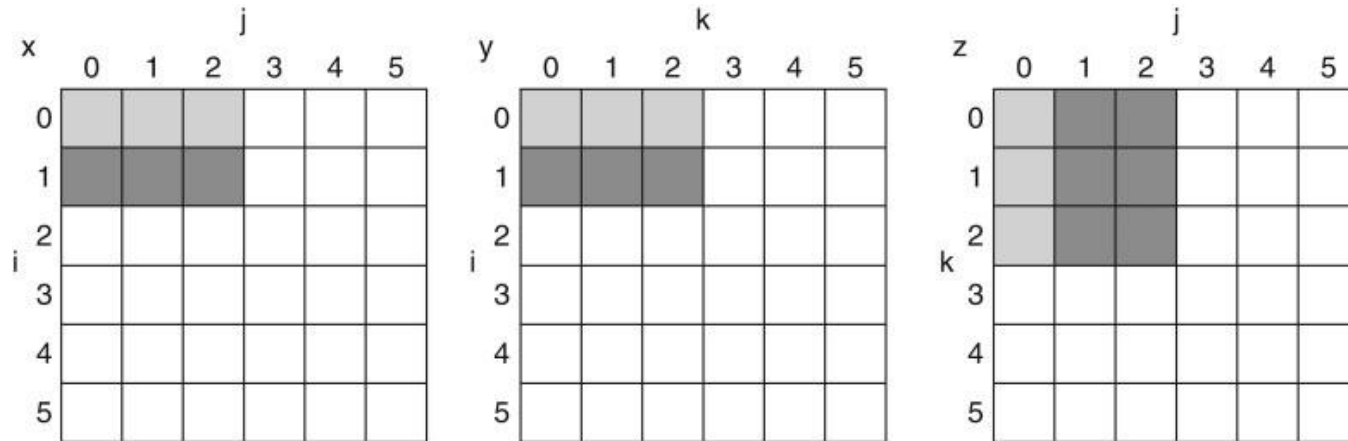


FIGURE 5.22 The age of accesses to the arrays C, A, and B when $BLOCKSIZE = 5 \times 3$. Note that, in contrast to Figure 5.20, fewer elements are accessed.

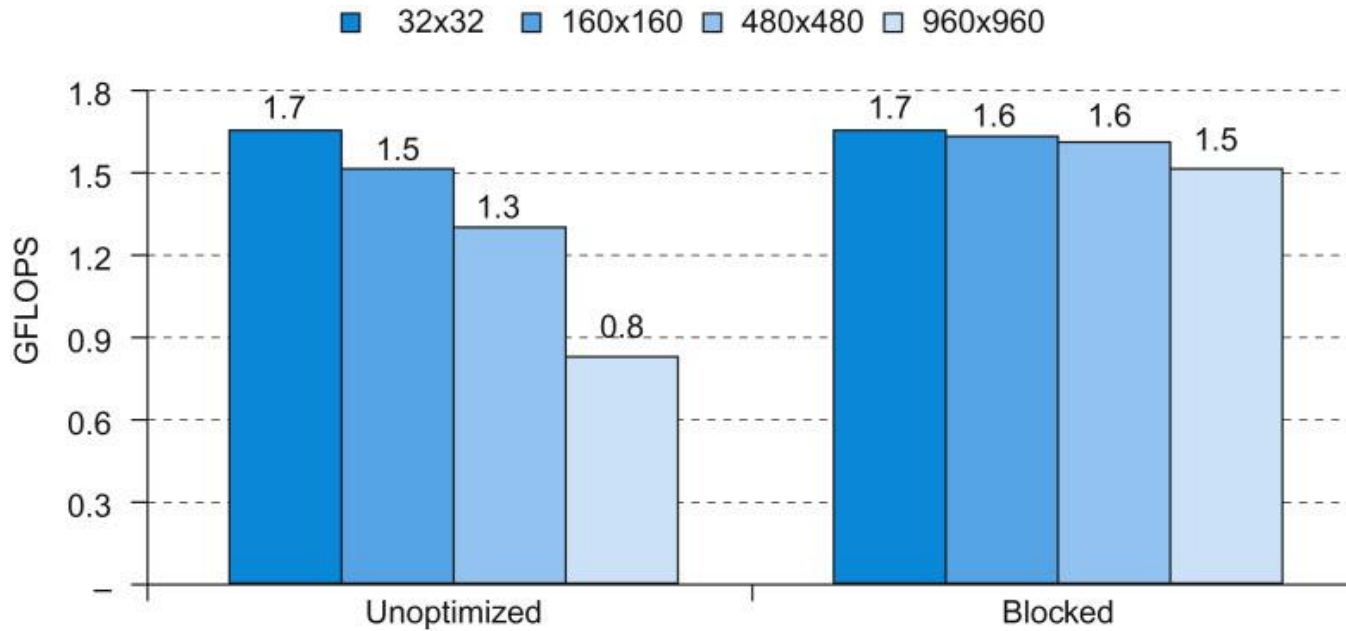


FIGURE 5.23 Performance of unoptimized DGEMM (Figure 3.21) versus cache blocked DGEMM (Figure 5.21) as the matrix dimension varies from 32x32 (where all three matrices fit in the cache) to 960x960.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

FIGURE 5.24 Parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits.

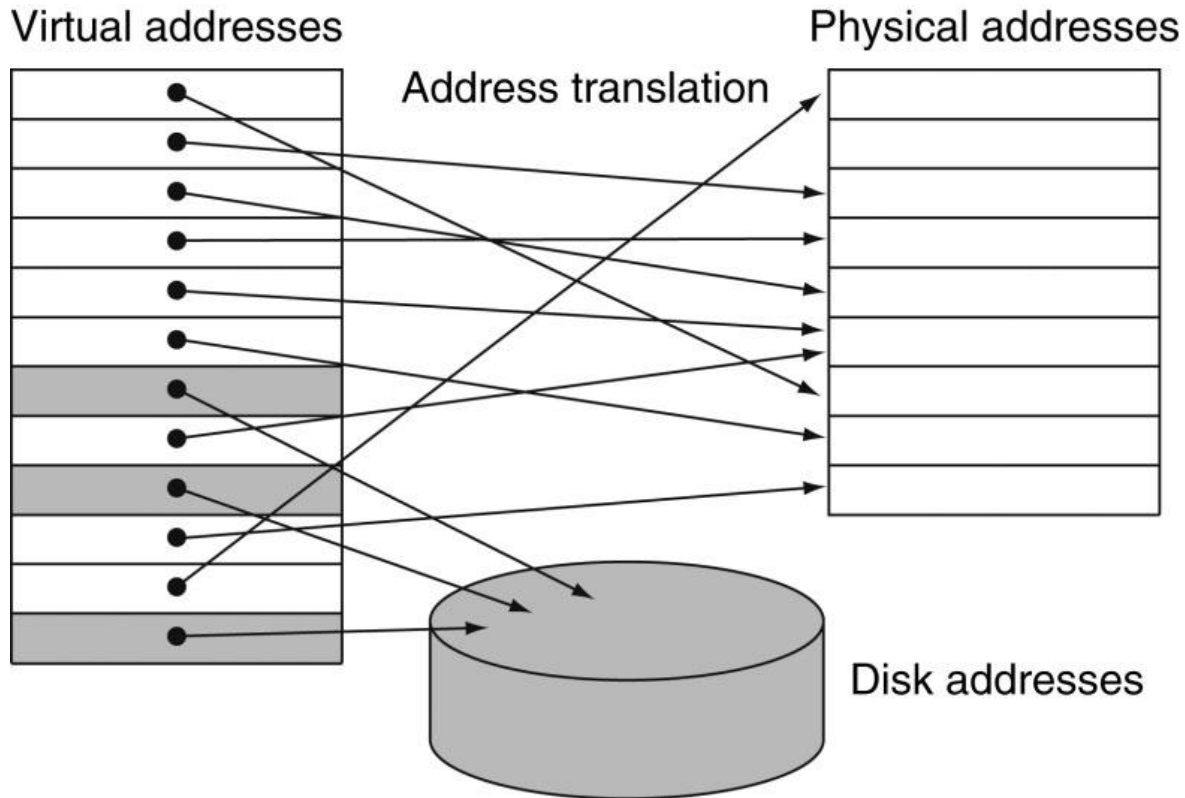


FIGURE 5.25 In virtual memory, blocks of memory (called *pages*) are mapped from one set of addresses (called *virtual addresses*) to another set (called *physical addresses*). The processor generates virtual addresses while the memory is accessed using physical addresses. Both the virtual memory and the physical memory are broken into pages, so that a virtual page is mapped to a physical page. Of course, it is also possible for a virtual page to be absent from main memory and not be mapped to a physical address; in that case, the page resides on disk. Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code.

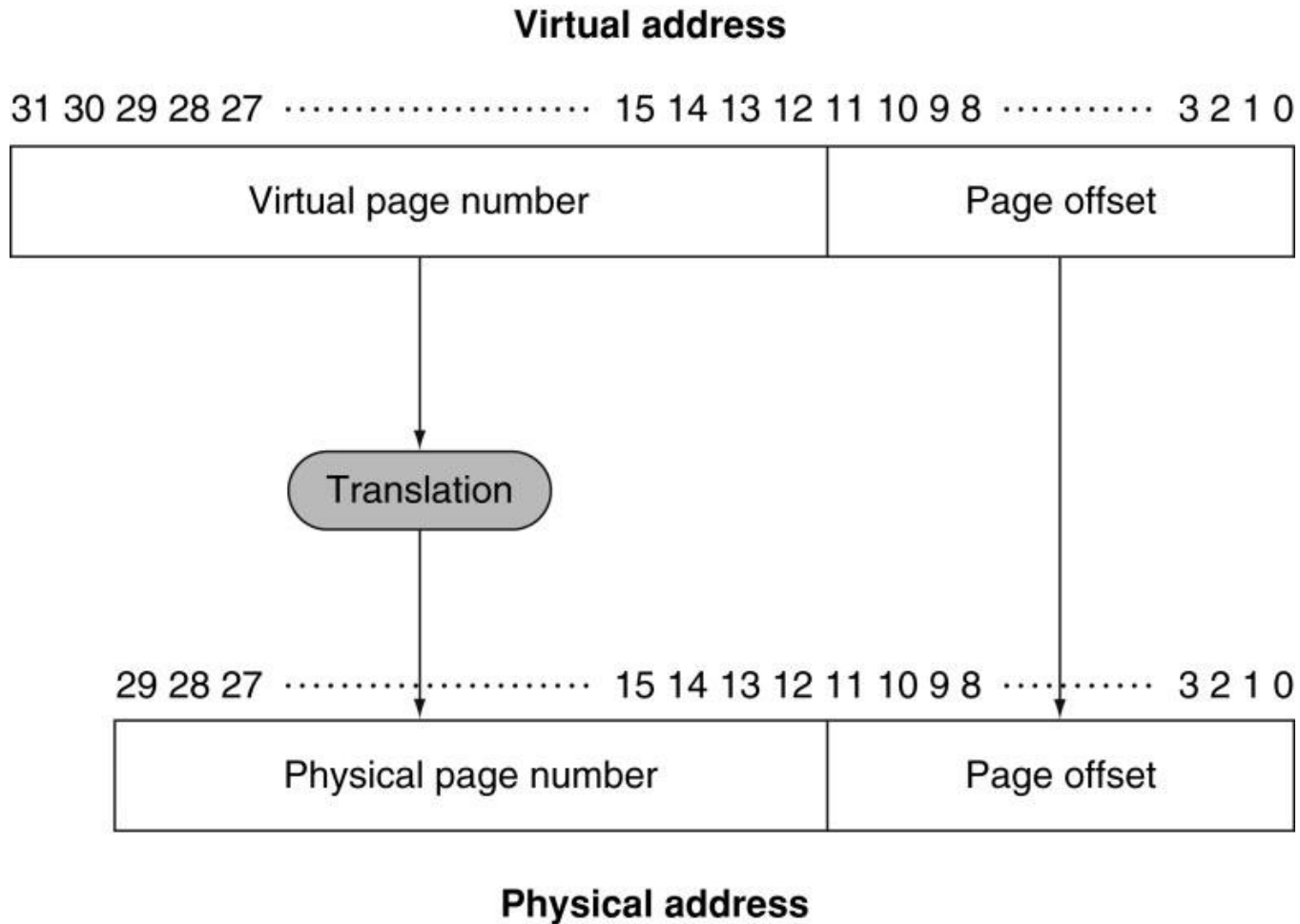


FIGURE 5.26 Mapping from a virtual to a physical address. The page size is $2^{12} \times 512$ KiB. The number of physical pages allowed in memory is 2^{18} , since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GiB, while the virtual address space is 4 GiB.

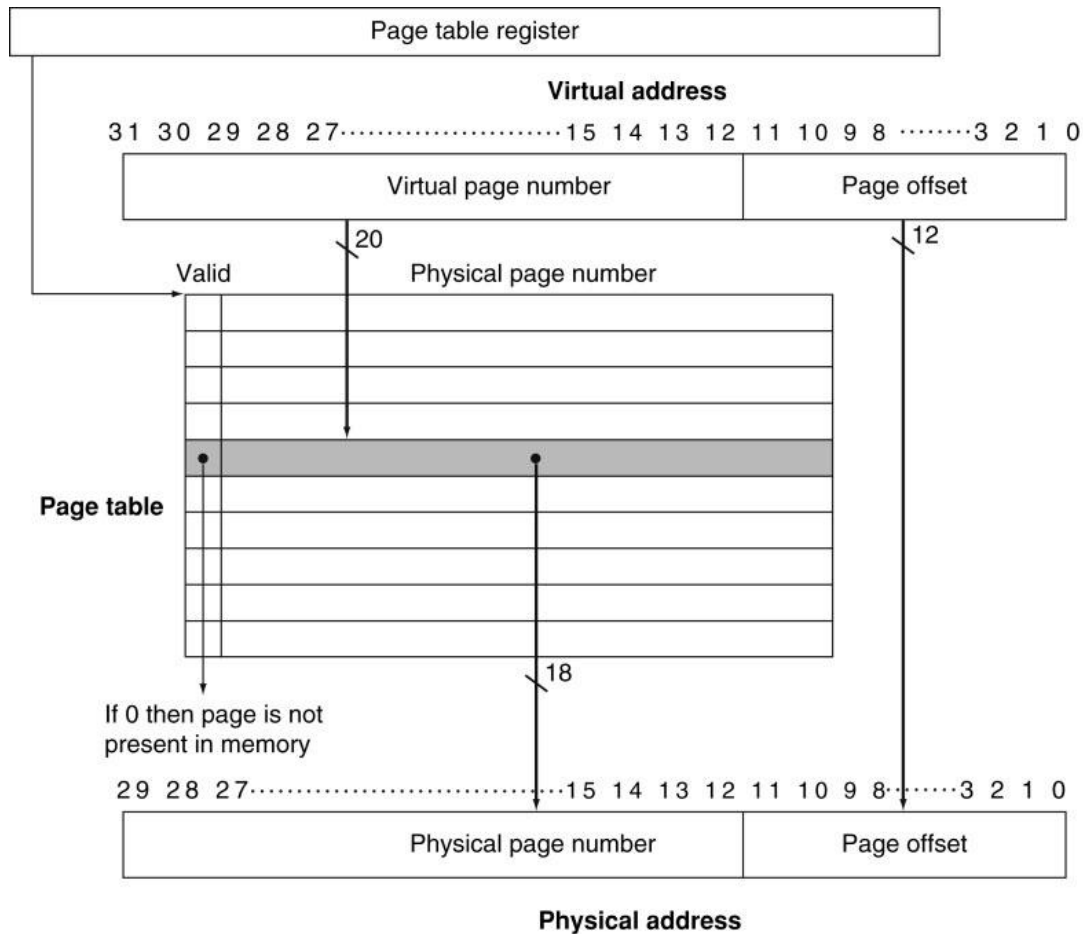


FIGURE 5.27 The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address. We assume a 32-bit address. The page table pointer gives the starting address of the page table. In this figure, the page size is 2^{12} bytes, or 4 KiB. The virtual address space is 2^{32} bytes, or 4 GiB, and the physical address space is 230 bytes, which allows main memory of up to 1 GiB. The number of entries in the page table is 2^{20} , or 1 million entries. The valid bit for each entry indicates whether the mapping is legal. If it is off, then the page is not present in memory. Although the page table entry shown here need only be 19 bits wide, it would typically be rounded up to 32 bits for ease of indexing. The extra bits would be used to store additional information that needs to be kept on a per-page basis, such as protection.

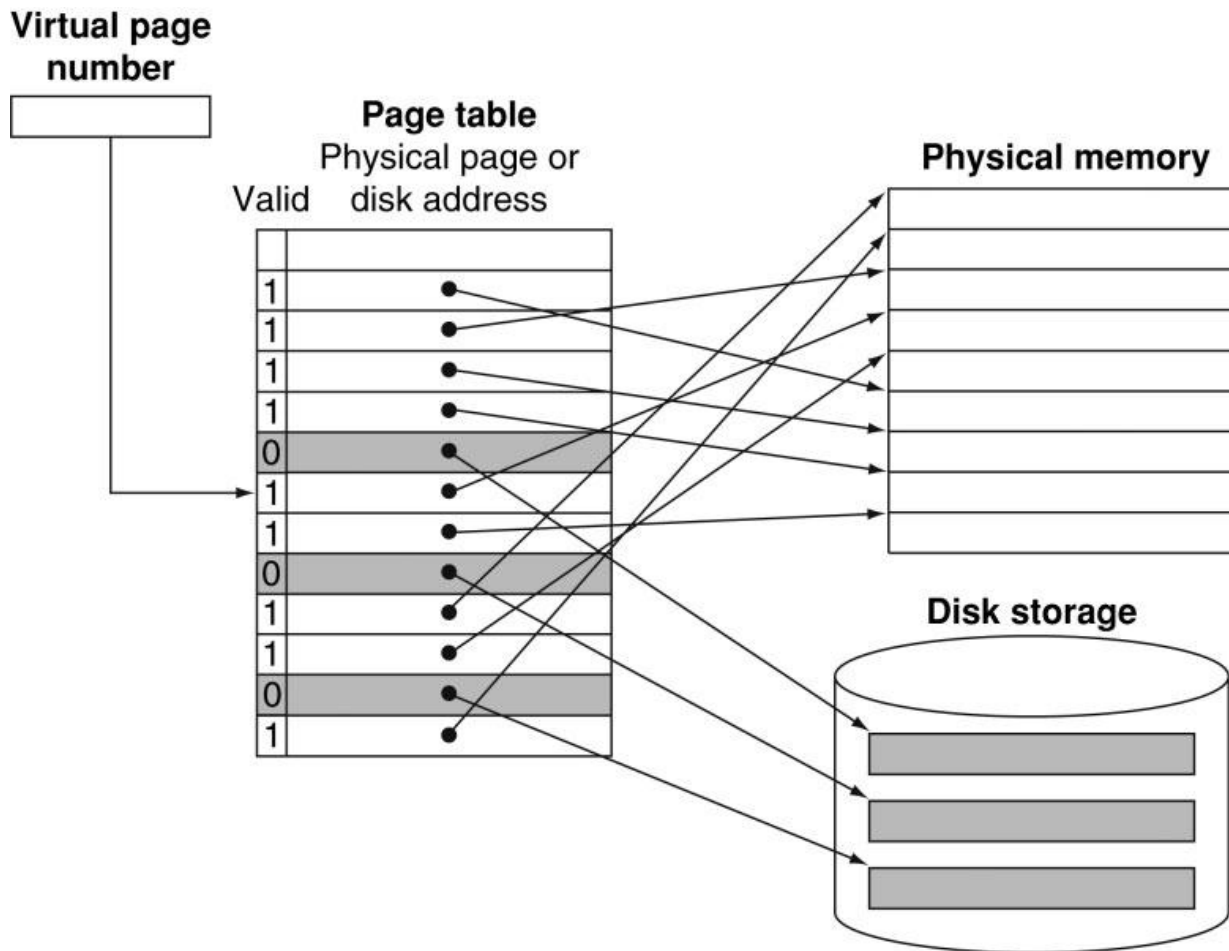


FIGURE 5.28 The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy. The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e., the starting address of the page in memory) corresponding to the virtual page. If the valid bit is off, the page currently resides only on disk, at a specified disk address. In many systems, the table of physical page addresses and disk page addresses, while logically one table, is stored in two separate data structures. Dual tables are justified in part because we must keep the disk addresses of all the pages, even if they are currently in main memory. Remember that the pages in main memory and the pages on disk are the same size.

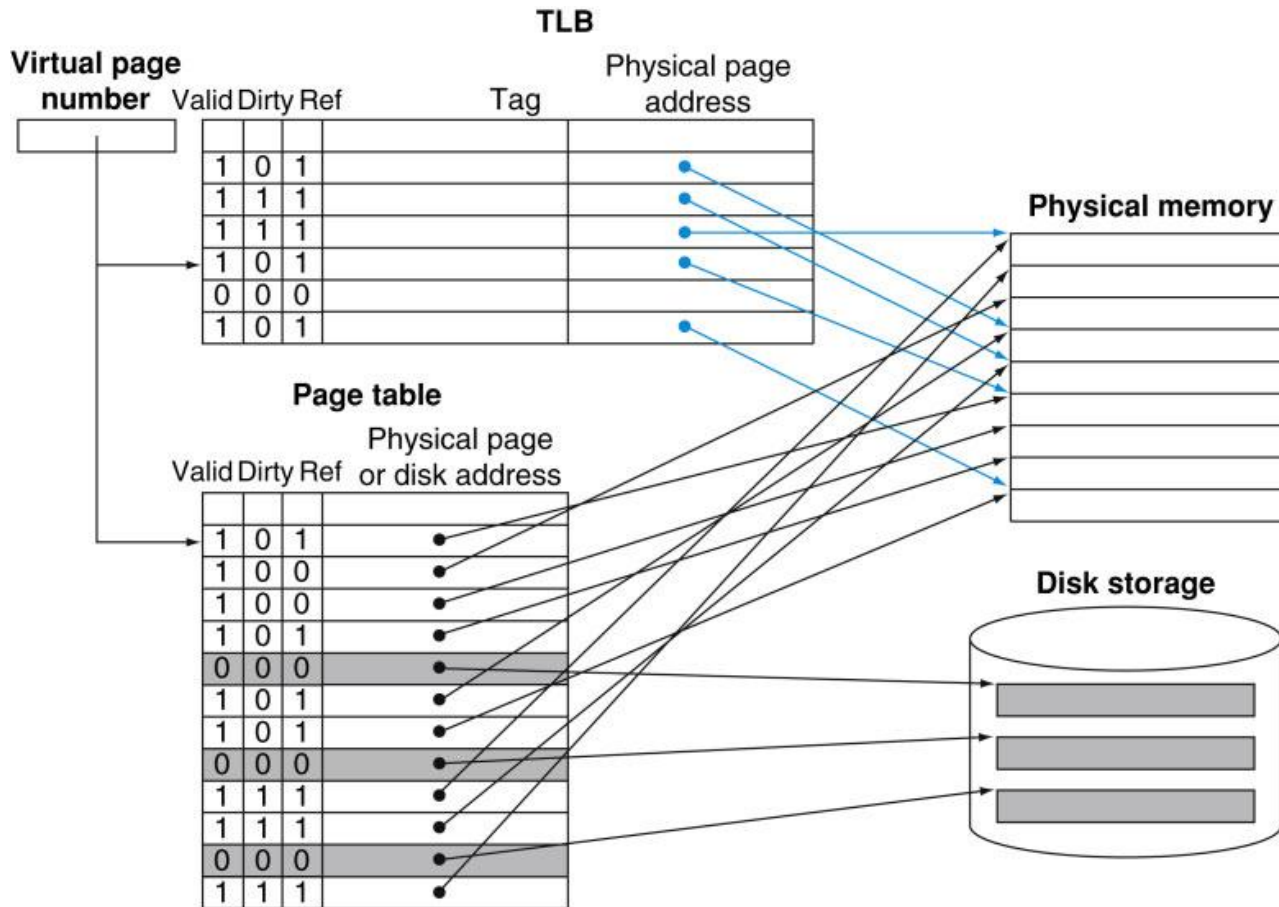


FIGURE 5.29 The TLB acts as a cache of the page table for the entries that map to physical pages only. The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. The TLB mappings are shown in color. Because the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. The page table either supplies a physical page number for the page (which can then be used to build a TLB entry) or indicates that the page resides on disk, in which case a page fault occurs. Since the page table has an entry for every virtual page, no tag field is needed; in other words, unlike a TLB, a page table is *not* a cache.

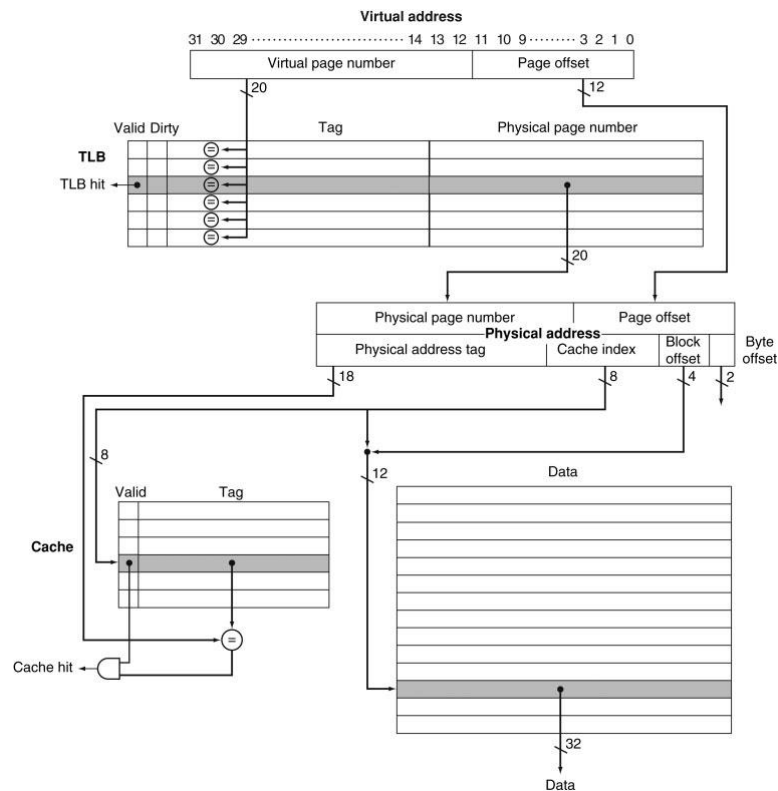


FIGURE 5.30 The TLB and cache implement the process of going from a virtual address to a data item in the Intrinsity FastMATH. This figure shows the organization of the TLB and the data cache, assuming a 4 KiB page size. This diagram focuses on a read; Figure 5.31 describes how to handle writes. Note that unlike Figure 5.12, the tag and data RAMs are split. By addressing the long but narrow data RAM with the cache index concatenated with the block offset, we select the desired word in the block without a 16:1 multiplexor. While the cache is direct mapped, the TLB is fully associative. Implementing a fully associative TLB requires that every TLB tag be compared against the virtual page number, since the entry of interest can be anywhere in the TLB. (See content addressable memories in the *Elaboration on* page 408.) If the valid bit of the matching entry is on, the access is a TLB hit, and bits from the physical page number together with bits from the page offset form the index that is used to access the cache.

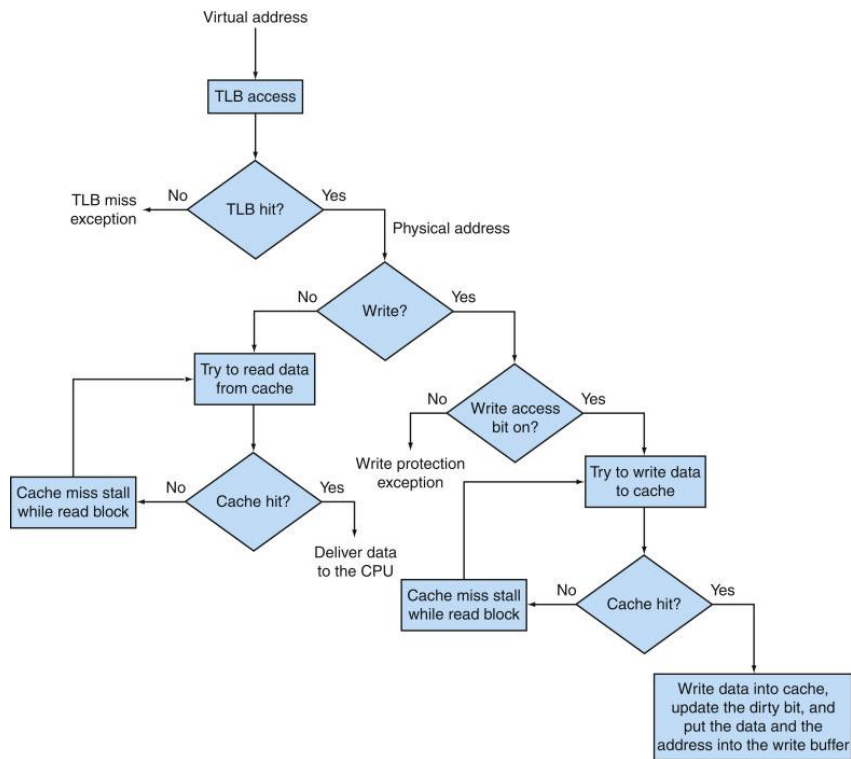


FIGURE 5.31 Processing a read or a write-through in the Intrinsity FastMATH TLB and cache. If the TLB generates a hit, the cache can be accessed with the resulting physical address. For a read, the cache generates a hit or miss and supplies the data or causes a stall while the data is brought from memory. If the operation is a write, a portion of the cache entry is overwritten for a hit and the data is sent to the write buffer if we assume write-through. A write miss is just like a read miss except that the block is modified after it is read from memory. Write-back requires writes to set a dirty bit for the cache block, and a write buffer is loaded with the whole block only on a read miss or write miss if the block to be replaced is dirty. Notice that a TLB hit and a cache hit are independent events, but a cache hit can only occur after a TLB hit occurs, which means that the data must be present in memory. The relationship between TLB misses and cache misses is examined further in the following example and the exercises at the end of this chapter.

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

FIGURE 5.32 The possible combinations of events in the TLB, virtual memory system, and cache. Three of these combinations are impossible, and one is possible (TLB hit, virtual memory hit, cache miss) but never detected.

Register	CP0 register number	Description
EPC	14	Where to restart after exception
Cause	13	Cause of exception
BadVAddr	8	Address that caused exception
Index	0	Location in TLB to be read or written
Random	1	Pseudorandom location in TLB
EntryLo	2	Physical page address and flags
EntryHi	10	Virtual page address
Context	4	Page table address and page number

FIGURE 5.33 MIPS control registers. These are considered to be in coprocessor 0, and hence are read using `mfc0` and written using `mtc0`.

Save state			
Save GPR	addi	\$k1,\$sp, -XCPSIZE	# save space on stack for state
	sw	\$sp, XCT_SP(\$k1)	# save \$sp on stack
	sw	\$v0, XCT_V0(\$k1)	# save \$v0 on stack
	...		# save \$v1, \$a1, \$s1, \$t1,... on stack
	sw	\$ra, XCT_RA(\$k1)	# save \$ra on stack
Save hi, lo	mfhi	\$v0	# copy Hi
	mflo	\$v1	# copy Lo
	sw	\$v0, XCT_HI(\$k1)	# save Hi value on stack
	sw	\$v1, XCT_LO(\$k1)	# save Lo value on stack
Save exception registers	mfc0	\$a0, \$scr	# copy cause register
	sw	\$a0, XCT_CR(\$k1)	# save \$scr value on stack
	...		# save \$v1,....
	mfc0	\$a3, \$sr	# copy status register
	sw	\$a3, XCT_SR(\$k1)	# save \$sr on stack
Set sp	move	\$sp, \$k1	# sp = sp - XCPSIZE
Enable nested exceptions			
	andi	\$v0, \$a3, MASK1	# \$v0 = \$sr & MASK1, enable exceptions
	mtc0	\$v0, \$sr	# \$sr = value that enables exceptions
Call C exception handler			
Set \$gp	move	\$gp, GPINIT	# set \$gp to point to heap area
Call C code	move	\$a0, \$sp	# arg1 = pointer to exception stack
	jal	xcpt_deliver	# call C code to handle exception
Restoring state			
Restore most GPR, hi, lo	move	\$at, \$sp	# temporary value of \$sp
	lw	\$ra, XCT_RA(\$at)	# restore \$ra from stack
	...		# restore \$t0,...., \$a1
	lw	\$a0, XCT_A0(\$k1)	# restore \$a0 from stack
Restore status register	lw	\$v0, XCT_SR(\$at)	# load old \$sr from stack
	li	\$v1, MASK2	# mask to disable exceptions
	and	\$v0, \$v0, \$v1	# \$v0 = \$sr & MASK2, disable exceptions
	mtc0	\$v0, \$sr	# set status register
Exception return			
Restore \$sp and rest of GPR used as temporary registers	lw	\$sp, XCT_SP(\$at)	# restore \$sp from stack
	lw	\$v0, XCT_V0(\$at)	# restore \$v0 from stack
	lw	\$v1, XCT_V1(\$at)	# restore \$v1 from stack
	lw	\$k1, XCT_EPC(\$at)	# copy old \$epc from stack
	lw	\$at, XCT_AT(\$at)	# restore \$at from stack
Restore ERC and return	mtc0	\$k1, \$epc	# restore \$epc
	eret	\$ra	# return to interrupted instruction

FIGURE 5.34 MIPS code to save and restore state on an exception.

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

FIGURE 5.35 The key quantitative design parameters that characterize the major elements of memory hierarchy in a computer. These are typical values for these levels as of 2012. Although the range of values is wide, this is partially because many of the values that have shifted over time are related; for example, as caches become larger to overcome larger miss penalties, block sizes also grow. While not shown, server microprocessors today also have L3 caches, which can be 2 to 8 MiB and contain many more blocks than L2 caches. L3 caches lower the L2 miss penalty to 30 to 40 clock cycles.

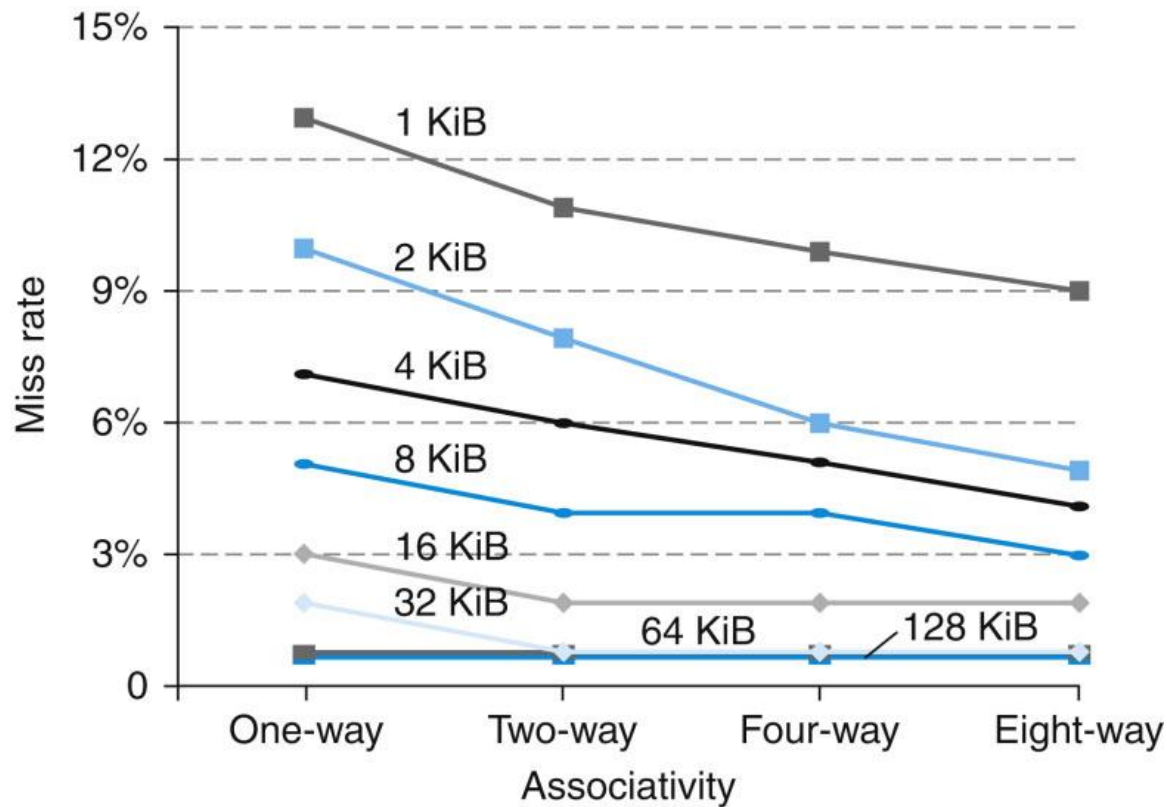


FIGURE 5.36 The data cache miss rates for each of eight cache sizes improve as the associativity increases. While the benefit of going from one-way (direct mapped) to two-way set associative is significant, the benefits of further associativity are smaller (e.g., 1%–10% improvement going from two-way to four-way versus 20%–30% improvement going from one-way to two-way). There is even less improvement in going from four-way to eight-way set associative, which, in turn, comes very close to the miss rates of a fully associative cache. Smaller caches obtain a significantly larger absolute benefit from associativity because the base miss rate of a small cache is larger. Figure 5.16 explains how this data was collected.

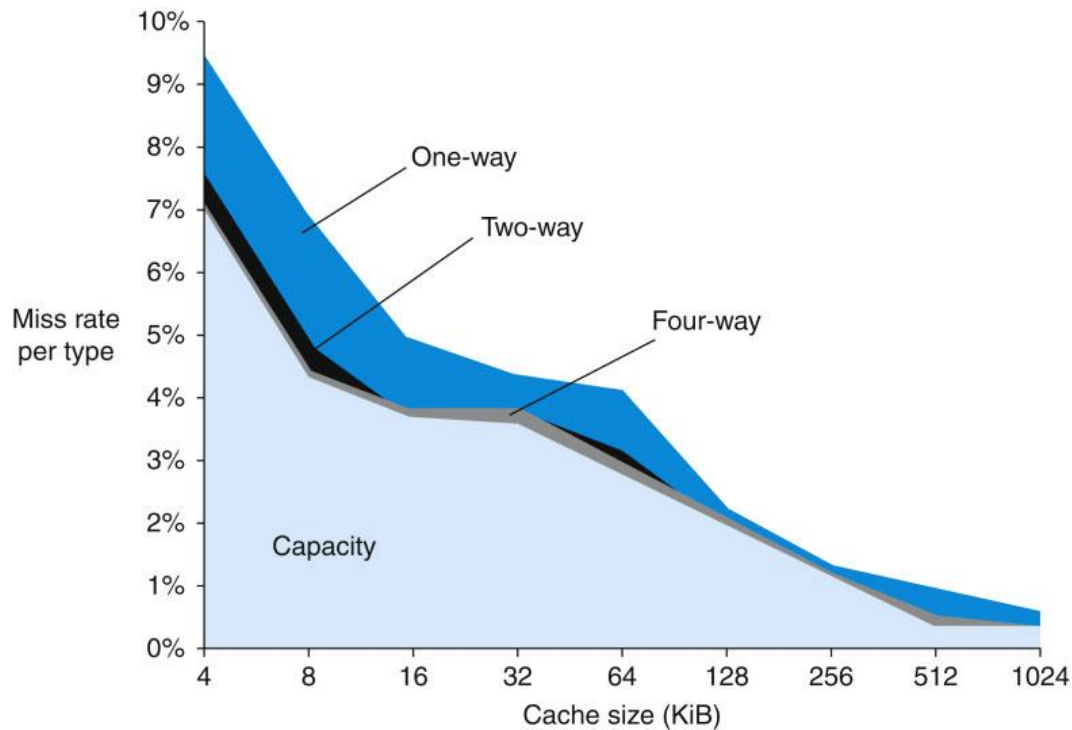


FIGURE 5.37 The miss rate can be broken into three sources of misses. This graph shows the total miss rate and its components for a range of cache sizes. This data is for the SPEC CPU2000 integer and floating-point benchmarks and is from the same source as the data in Figure 5.36 The compulsory miss component is 0.006% and cannot be seen in this graph. The next component is the capacity miss rate, which depends on cache size. The conflict portion, which depends both on associativity and on cache size, is shown for a range of associativities from one-way to eight-way. In each case, the labeled section corresponds to the increase in the miss rate that occurs when the associativity is changed from the next higher degree to the labeled degree of associativity. For example, the section labeled *two-way* indicates the additional misses arising when the cache has associativity of two rather than four. Thus, the difference in the miss rate incurred by a direct-mapped cache versus a fully associative cache of the same size is given by the sum of the sections marked *four-way*, *two-way*, and *one-way*. The difference between eight-way and four-way is so small that it is difficult to see on this graph.

Design change	Effect on miss rate	Possible negative performance effect
Increases cache size	Decreases capacity misses	May increase access time
Increases associativity	Decreases miss rate due to conflict misses	May increase access time
Increases block size	Decreases miss rate for a wide range of block sizes due to spatial locality	Increases miss penalty. Very large block could increase miss rate

FIGURE 5.38 Memory hierarchy design challenges.

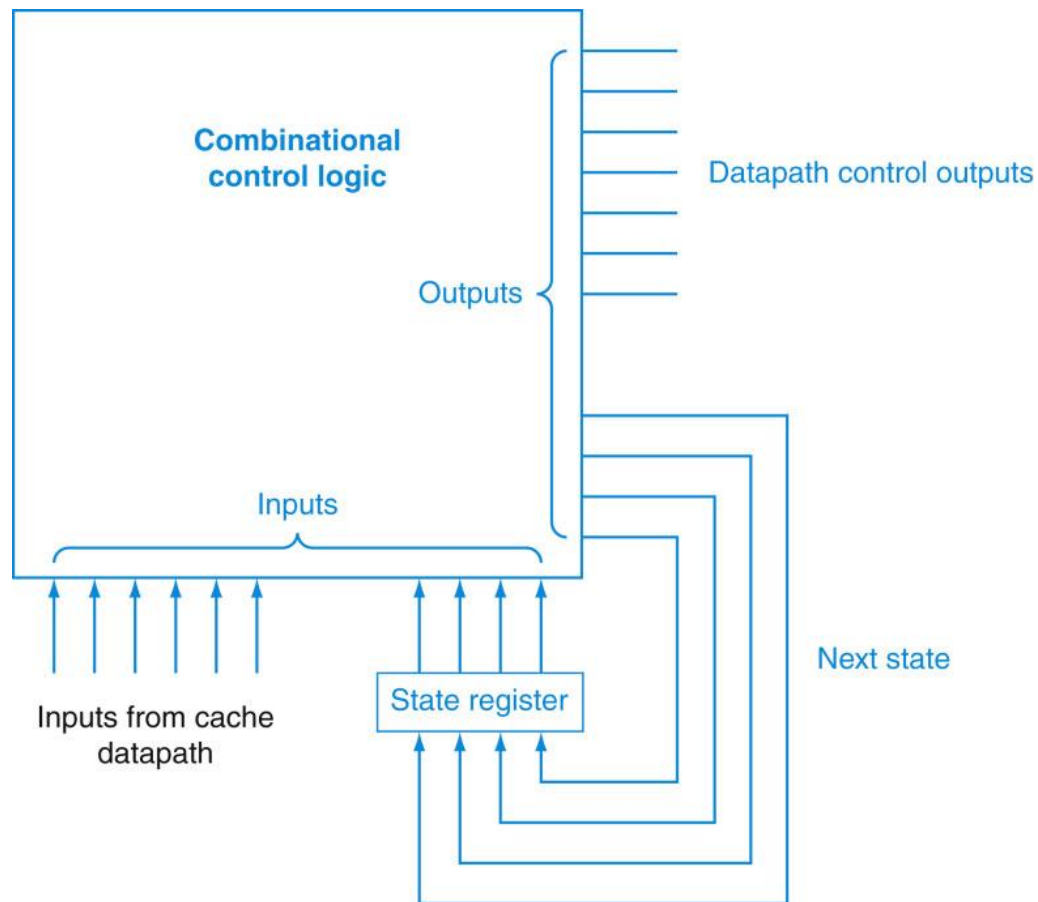


FIGURE 5.39 Finite-state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state. The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational logic are the current state and any inputs used to determine the next state. Notice that in the finite-state machine used in this chapter, the outputs depend only on the current state, not on the inputs. The *Elaboration* explains this in more detail.

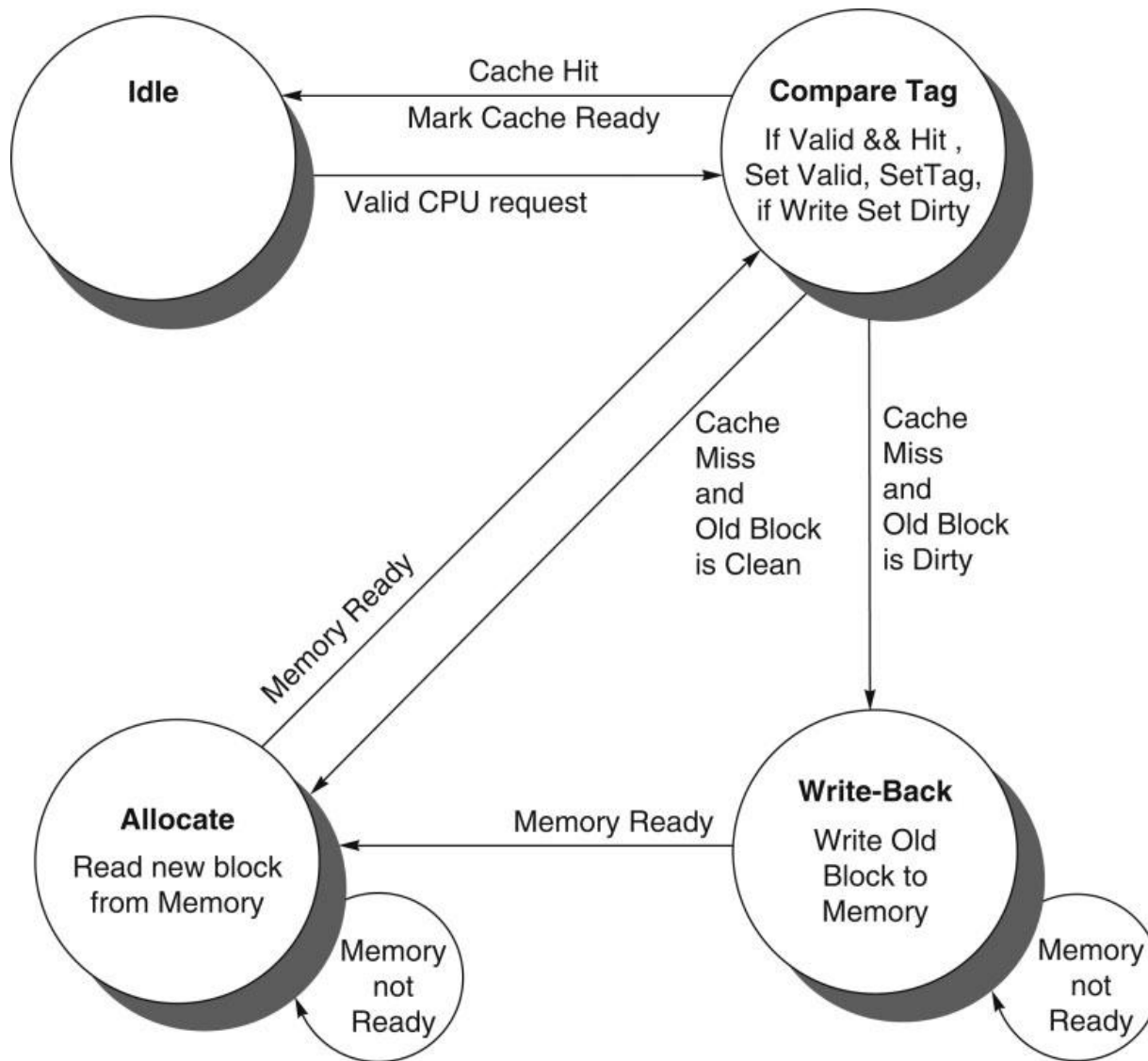


FIGURE 5.40 Four states of the simple controller.

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

FIGURE 5.41 The cache coherence problem for a single memory location (X), read and written by two processors (A and B). We initially assume that neither cache contains the variable and that X has the value 0. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X, it will receive 0!

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

FIGURE 5.42 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The CPU and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, CPU A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write-back only if the block is replaced. This requires the introduction of an additional state called "owner," which indicates that a block may be shared, but the owning processor is responsible for updating any other processors and memory when it changes the block or replaces it.

Characteristic	ARM Cortex-A8	Intel Core i7
Virtual address	32 bits	48 bits
Physical address	32 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 16 MiB	Variable: 4 KiB, 2/4 MiB
TLB organization	<p>1 TLB for instructions and 1 TLB for data</p> <p>Both TLBs are fully associative, with 32 entries, round robin replacement</p> <p>TLB misses handled in hardware</p>	<p>1 TLB for instructions and 1 TLB for data per core</p> <p>Both L1 TLBs are four-way set associative, LRU replacement</p> <p>L1 I-TLB has 128 entries for small pages, 7 per thread for large pages</p> <p>L1 D-TLB has 64 entries for small pages, 32 for large pages</p> <p>The L2 TLB is four-way set associative, LRU replacement</p> <p>The L2 TLB has 512 entries</p> <p>TLB misses handled in hardware</p>

FIGURE 5.43 Address translation and TLB hardware for the ARM Cortex-A8 and Intel Core i7 920. Both processors provide support for large pages, which are used for things like the operating system or mapping a frame buffer. The large-page scheme avoids using a large number of entries to map a single object that is always present.

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	-	Unified (instruction and data)
L3 cache size	-	8 MiB, shared
L3 cache associativity	-	16-way set associative
L3 replacement	-	Approximated LRU
L3 block size	-	64 bytes
L3 write policy	-	Write-back, Write-allocate
L3 hit time	-	35 clock cycles

FIGURE 5.44 Caches in the ARM Cortex-A8 and Intel Core i7 920.

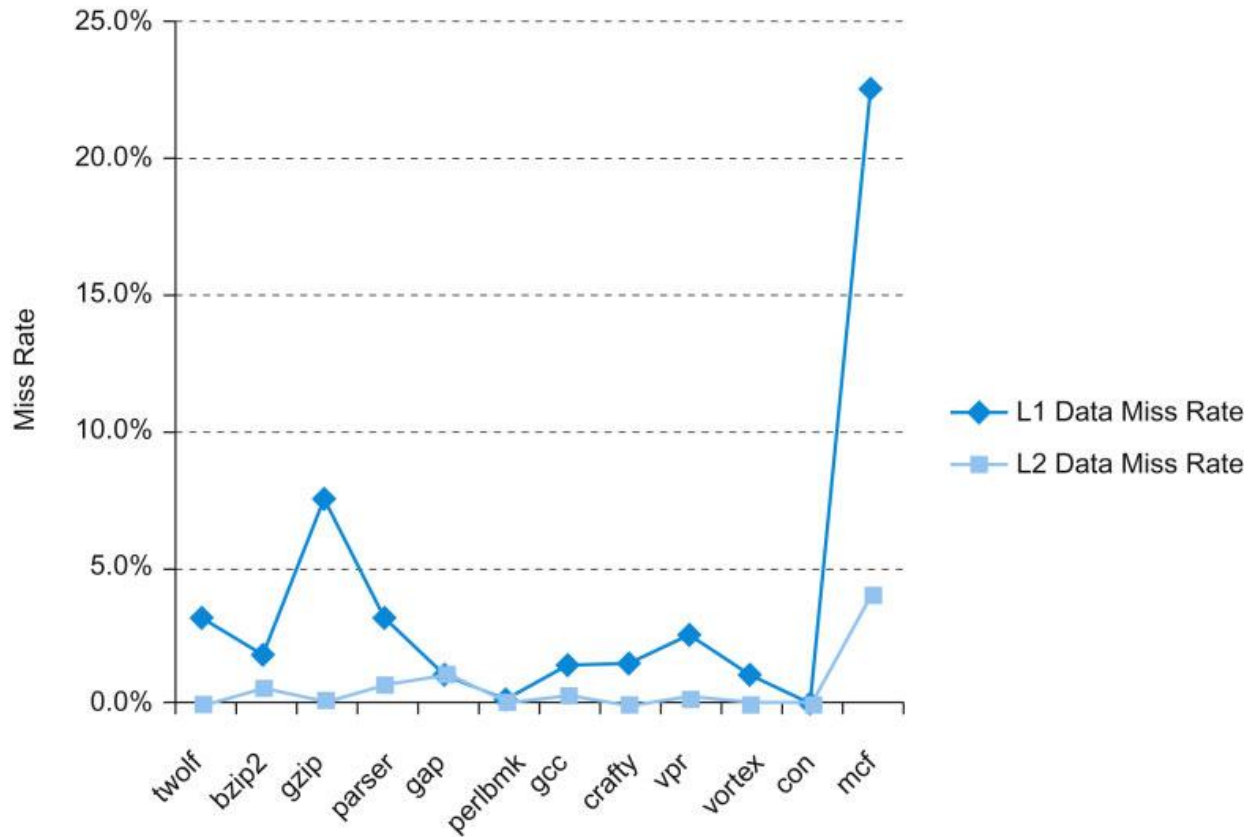


FIGURE 5.45 Data cache miss rates for ARM Cortex-A8 when running Minnespec, a small version of SPEC2000. Applications with larger memory footprints tend to have higher miss rates in both L1 and L2. Note that the L2 rate is the global miss rate; that is, counting all references, including those that hit in L1. (See Elaboration in Section 5.4.) Mcf is known as a cache buster. Note that this figure is for the same systems and benchmarks as Figure 4.76 in Chapter 4.

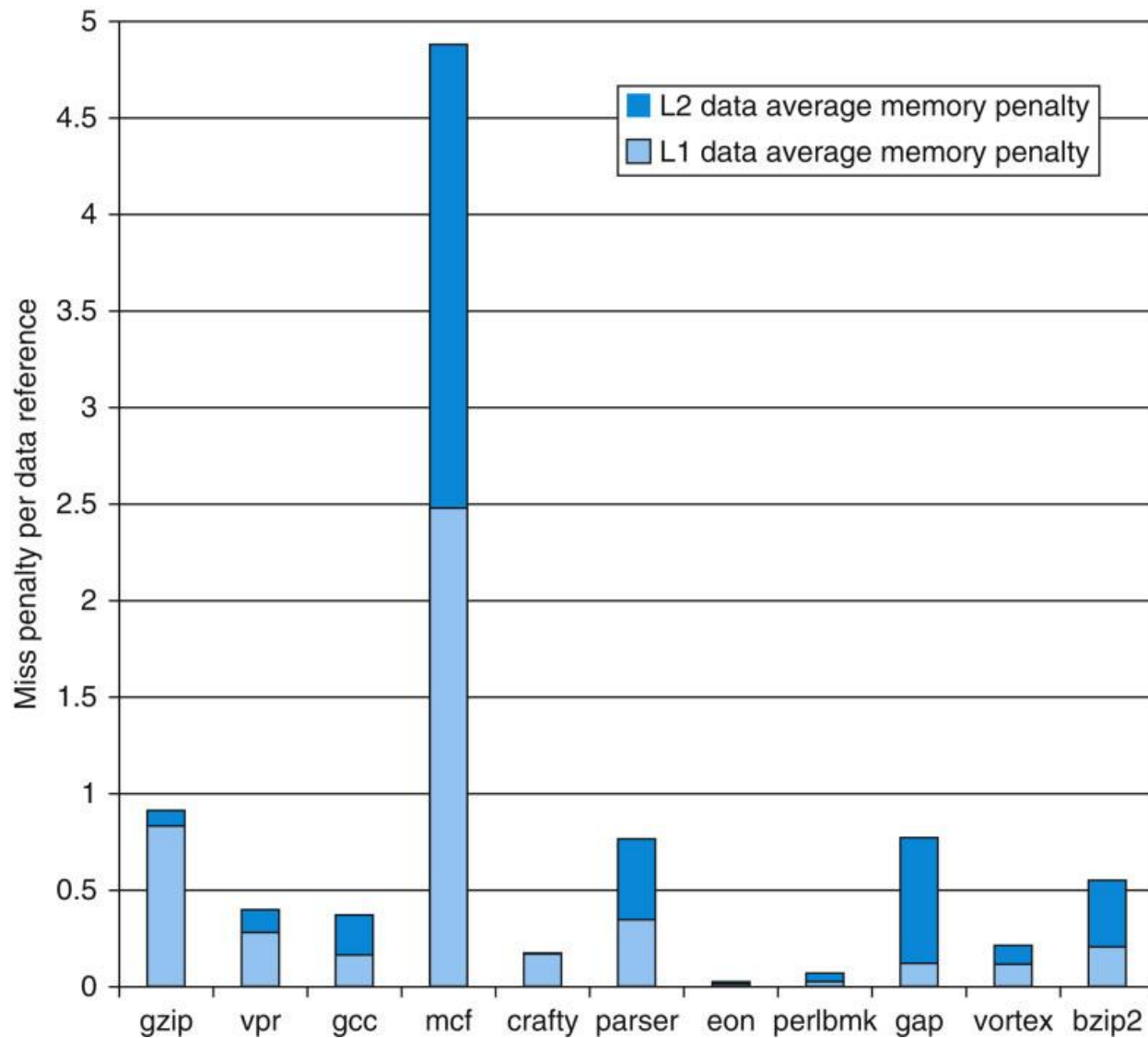


FIGURE 5.46 The average memory access penalty in clock cycles per data memory reference coming from L1 and L2 is shown for the ARM processor when running Minnespec. Although the miss rates for L1 are significantly higher, the L2 miss penalty, which is more than five times higher, means that the L2 misses can contribute significantly.

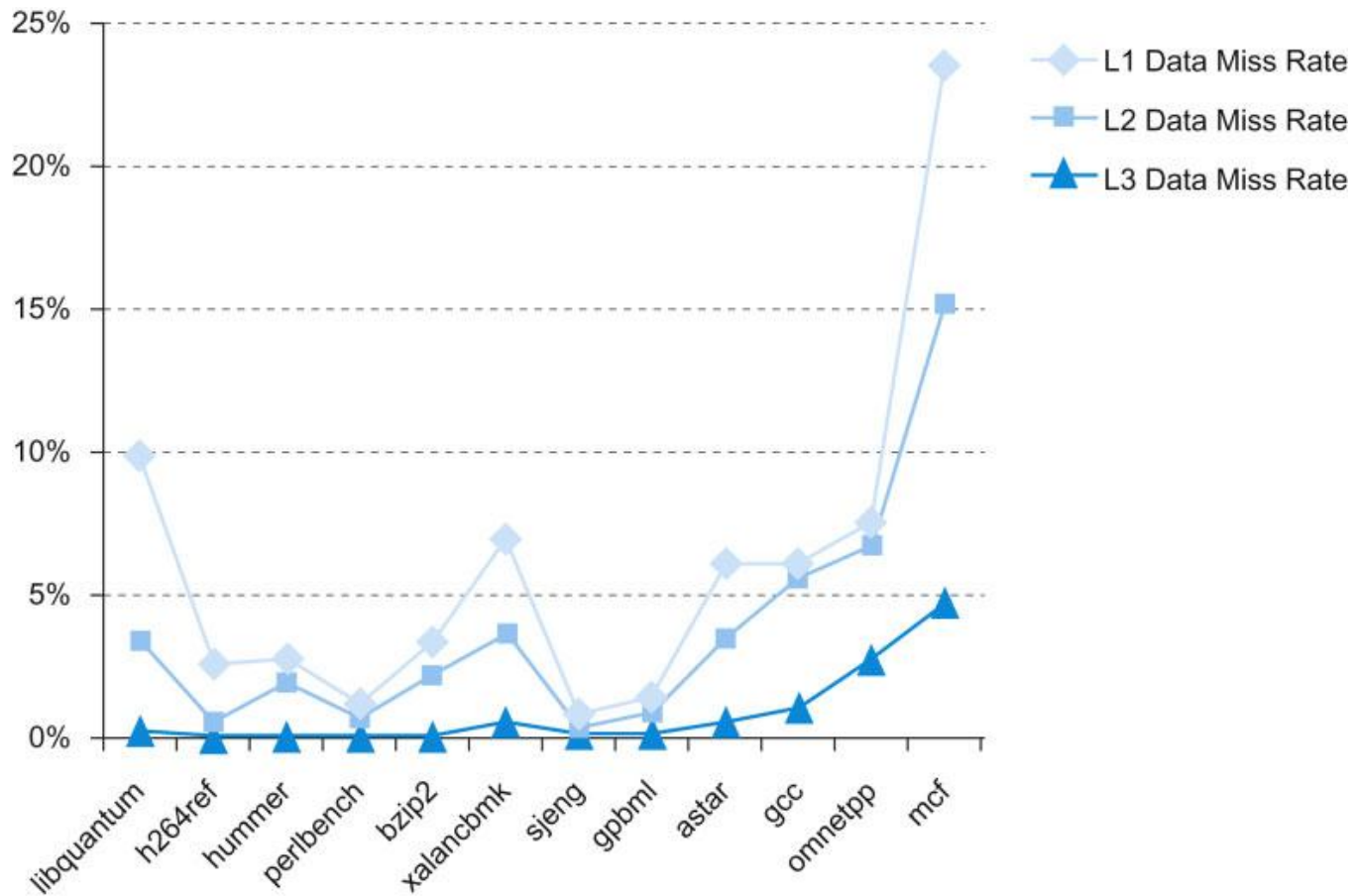


FIGURE 5.47 The L1, L2, and L3 data cache miss rates for the Intel Core i7 920 running the full integer SPEC CPU2006 benchmarks.


```

1 #include <x86intrin.h>
2 #define UNROLL (4)
3 #define BLOCKSIZE 32
4 void do_block (int n, int si, int sj, int sk,
5               double *A, double *B, double *C)
6 {
7     for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
8         for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
9             __m256d c[4];
10            for ( int x = 0; x < UNROLL; x++ )
11                c[x] = _mm256_load_pd(C+i+x*4+j*n);
12            /* c[x] = C[i][j] */
13            for( int k = sk; k < sk+BLOCKSIZE; k++ )
14                {
15                __m256d b = _mm256_broadcast_sd(B+k+j*n);
16                /* b = B[k][j] */
17                for (int x = 0; x < UNROLL; x++)
18                    c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                                     _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20                }
21
22
23            for ( int x = 0; x < UNROLL; x++ )
24                _mm256_store_pd(C+i+x*4+j*n, c[x]);
25                /* C[i][j] = c[x] */
26        }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30     for ( int sj = 0; sj < n; sj += BLOCKSIZE )
31         for ( int si = 0; si < n; si += BLOCKSIZE )
32             for ( int sk = 0; sk < n; sk += BLOCKSIZE )
33                 do_block(n, si, sj, sk, A, B, C);
34 }

```

FIGURE 5.48 Optimized C version of DGEMM from Figure 4.80 using cache blocking. These changes are the same ones found in Figure 5.21. The assembly language produced by the compiler for the `do_block` function is nearly identical to Figure 4.81. Once again, there is no overhead to call the `do_block` because the compiler inlines the function call.

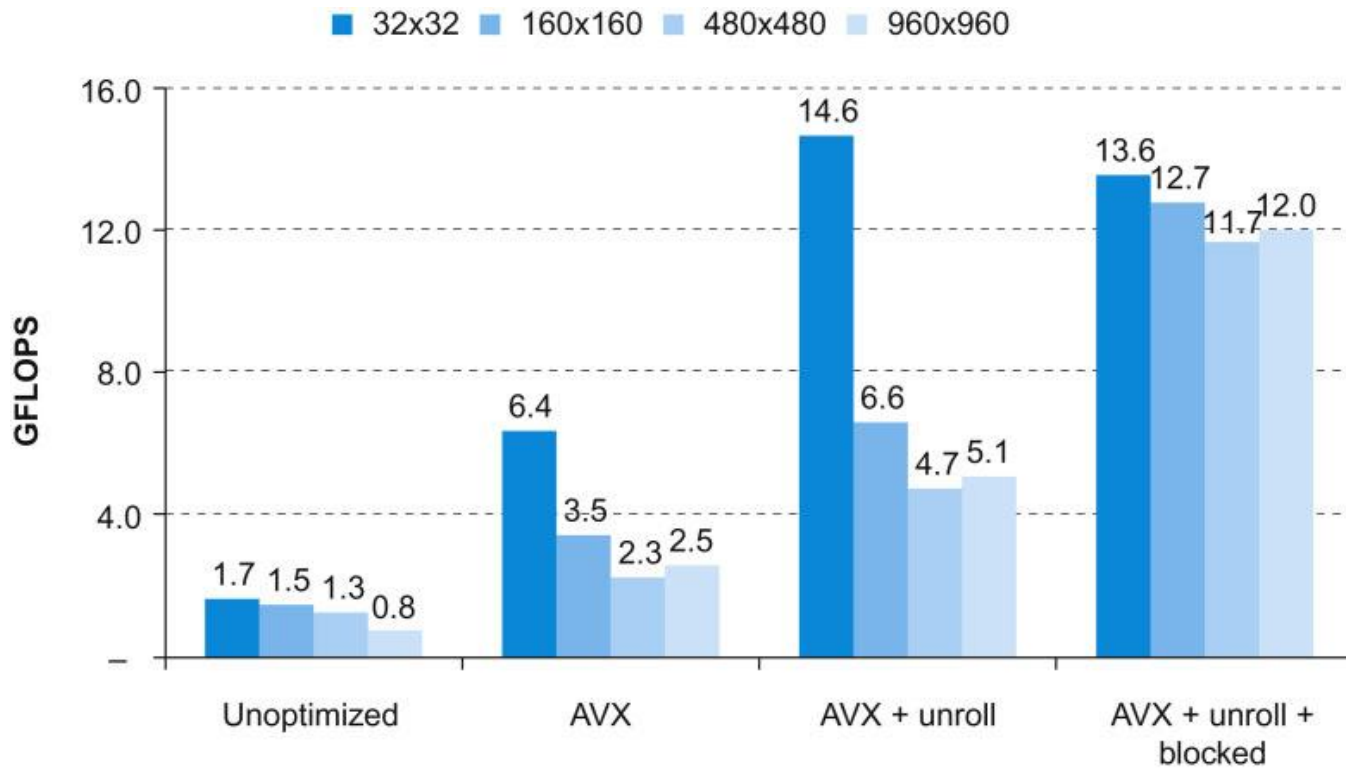


FIGURE 5.49 Performance of four versions of DGEMM from matrix dimensions 32x32 to 960x960. The fully optimized code for largest matrix is almost 15 times as fast the unoptimized version in Figure 3.21 in Chapter 3.

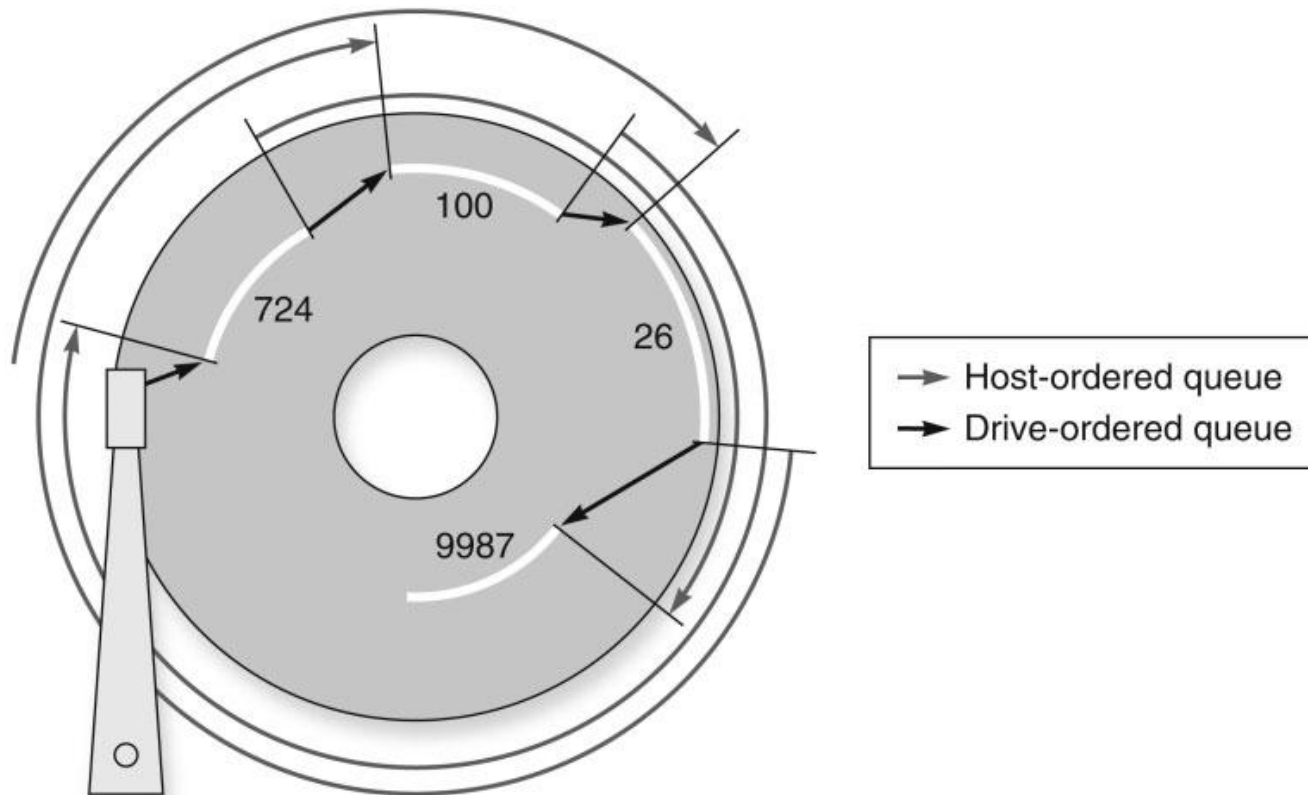
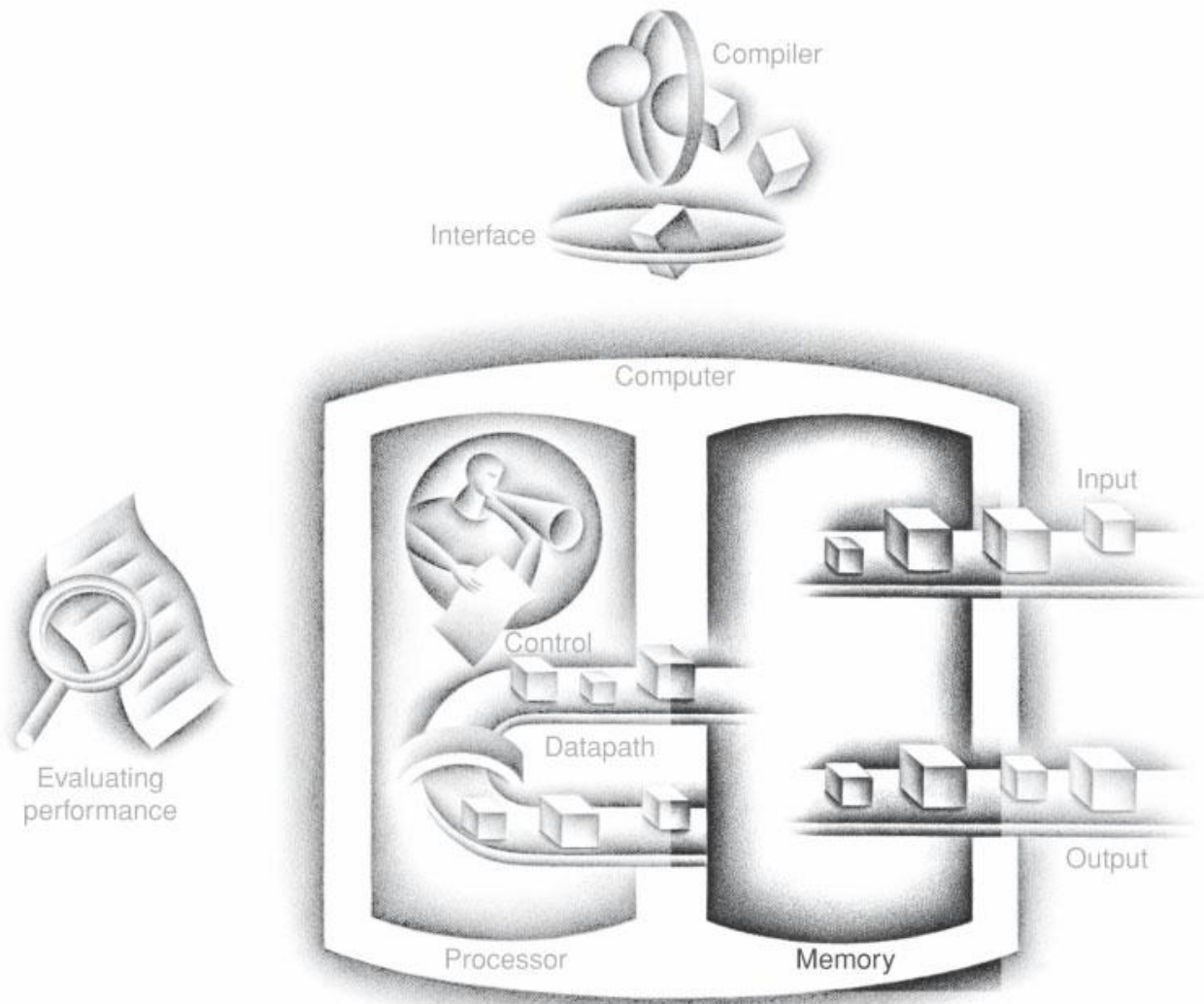


FIGURE 5.50 Example showing OS versus disk schedule accesses, labeled host-ordered versus drive-ordered. The former takes three revolutions to complete the four reads, while the latter completes them in just three-fourths of a revolution (from Anderson [2003]).

Problem category	Problem x86 instructions
Access sensitive registers without trapping when running in user mode	Store global descriptor table register (SGDT) Store local descriptor table register (SLDT) Store interrupt descriptor table register (SIDT) Store machine status word (SMSW) Push flags (PUSHF, PUSHFD) Pop flags (POPF, POPFD)
When accessing virtual memory mechanisms in user mode, instructions fail the x86 protection checks	Load access rights from segment descriptor (LAR) Load segment limit from segment descriptor (LSL) Verify if segment descriptor is readable (VERR) Verify if segment descriptor is writable (VERW) Pop to segment register (POP CS, POP SS, . . .) Push segment register (PUSH CS, PUSH SS, . . .) Far call to different privilege level (CALL) Far return to different privilege level (RET) Far jump to different privilege level (JMP) Software interrupt (INT) Store segment selector register (STR) Move to/from segment registers (MOVE)

FIGURE 5.51 Summary of 18 x86 instructions that cause problems for virtualization [Robin and Irvine, 2000]. The first five instructions in the top group allow a program in user mode to read a control register, such as descriptor table registers, without causing a trap. The pop flags instruction modifies a control register with sensitive information but fails silently when in user mode. The protection checking of the segmented architecture of the x86 is the downfall of the bottom group, as each of these instructions checks the privilege level implicitly as part of instruction execution when reading a control register. The checking assumes that the OS must be at the highest privilege level, which is not the case for guest VMs. Only the Move to segment register tries to modify control state, and protection checking foils it as well.



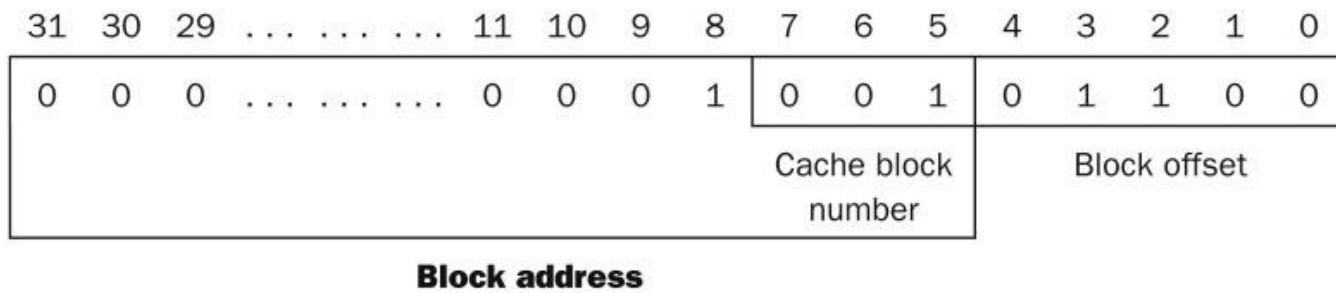
Unn Fig. 1

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

Unn Fig. 2

```
TLBmiss:
    mfc0    $k1,Context    # copy address of PTE into temp $k1
    lw      $k1,0($k1)     # put PTE into temp $k1
    mtc0    $k1,EntryLo    # put PTE into special register EntryLo
    tlbwr                               # put EntryLo into TLB entry at Random
    eret                               # return from TLB miss exception
```

Unn Fig. 3



Unn Fig. 4


```
for I=1:8
    for J=1:8000
        A(I,J)=B(I,0)+A(J,I);
    end
end
```

Unn Fig. 5

```
struct entry {  
    int srcIP;    // remote IP address  
    char URL[128]; // request URL (e.g., "GET index.html")  
    long long refTime; // reference time  
    int status; // connection status  
    char browser[64]; // client browser name  
} log [NUM_ENTRIES];
```

Unn Fig. 6

Memory technology	Typical access time	\$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	\$500–\$1000
DRAM semiconductor memory	50–70 ns	\$10–\$20
Flash semiconductor memory	5,000–50,000 ns	\$0.75–\$1.00
Magnetic disk	5,000,000–20,000,000 ns	\$0.05–\$0.10

Table 1

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.6b)	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss (5.6c)	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss (5.6d)	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss (5.6e)	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss (5.6f)	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Table 2

Block address	Cache block
0	$(0 \text{ modulo } 4) = 0$
6	$(6 \text{ modulo } 4) = 2$
8	$(8 \text{ modulo } 4) = 0$

Table 3

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Table 4

Block address	Cache set
0	$(0 \text{ modulo } 2) = 0$
6	$(6 \text{ modulo } 2) = 0$
8	$(8 \text{ modulo } 2) = 0$

Table 5

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Table 6

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

Table 7

- | | |
|----------------|-----------------------------------|
| 1. L1 cache | a. A cache for a cache |
| 2. L2 cache | b. A cache for disks |
| 3. Main memory | c. A cache for a main memory |
| 4. TLB | d. A cache for page table entries |

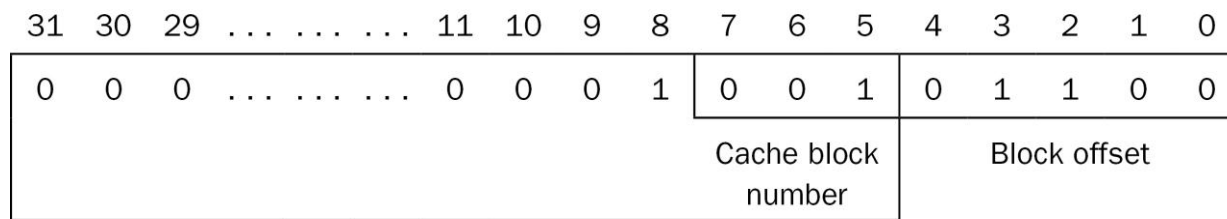
Table 8

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

Table 9

Associativity	Location method	Comparisons required
Direct mapped	Index	1
Set associative	Index the set, search among elements	Degree of associativity
Full	Search all cache entries	Size of the cache
	Separate lookup table	0

Table 10



Block address

Table 11

Operation	Starting LBA	Length
Read	724	8
Read	100	16
Read	9987	1
Read	26	128

Table 12

Operation	Starting LBA	Length
Read	26	128
Read	100	16
Read	724	8
Read	9987	1

Table 13

```
for (I=0; I<8; I++)  
    for (J=0; J<8000; J++)  
        A[I][J]=B[I][0]+A[J][I];
```

Table 14


```
for I=1:8
    for J=1:8000
        A(I,J)=B(I,0)+A(J,I);
    end
end
```

Table 15

3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253

Table 16

Tag	Index	Offset
31–10	9–5	4–0

Table 17

Address											
0	4	16	132	232	160	1024	30	140	3100	180	2180

Table 18

L1	L2
Write through, non-write allocate	Write back, write allocate

Table 19

Data Reads per 1000 Instructions	Data Writes per 1000 Instructions	Instruction Cache Miss Rate	Data Cache Miss Rate	Block Size (byte)
250	100	0.30%	2%	64

Table 20

0, 2, 4, 6, 8, 10, 12, 14, 16, ...

Table 21

8: 4%	16: 3%	32: 2%	64: 1.5%	128: 1%
-------	--------	--------	----------	---------

Table 22

	L1 Size	L1 Miss Rate	L1 Hit Time
P1	2 KiB	8.0%	0.66 ns
P2	4 KiB	6.0%	0.90 ns

Table 23

L2 Size	L2 Miss Rate	L2 Hit Time
1 MiB	95%	5.62 ns

Table 24

Base CPI, No Memory Stalls	Processor Speed	Main Memory Access Time	First Level Cache MissRate per Instruction	Second Level Cache, Direct-Mapped Speed	Global Miss Rate with Second Level Cache, Direct-Mapped	Second Level Cache, Eight-Way Set Associative Speed	Global Miss Rate with Second Level Cache, Eight-Way Set Associative
1.5	2 GHz	100 ns	7%	12 cycles	3.5%	28 cycles	1.5%

Table 25

MTTF	MTTR
3 Years	1 Day

Table 26

Page Size (KiB)	Page Utility or B-Tree Depth (Number of Disk Accesses Saved)	Index Page Access Cost (ms)	Utility/Cost
2	6.49 (or $\log_2(2048/16 \times 0.7)$)	10.2	0.64
4	7.49	10.4	0.72
8	8.49	10.8	0.79
16	9.49	11.6	0.82
32	10.49	13.2	0.79
64	11.49	16.4	0.70
128	12.49	22.8	0.55
256	13.49	35.6	0.38

Table 27

Year	DRAM Cost (\$/MiB)	Page Size (KiB)	Disk Cost (\$/disk)	Disk Access Rate (access/sec)
1987	5000	1	15,000	15
1997	15	8	2000	64
2007	0.05	64	80	83

Table 28

4669, 2227, 13916, 34587, 48870, 12608, 49225

Table 29

Valid	Tag	Physical Page Number
1	11	12
1	7	4
1	3	6
0	4	9

Table 30

Valid	Physical Page or in Disk
1	5
0	Disk
0	Disk
1	6
1	9
1	11
0	Disk
1	4
0	Disk
0	Disk
1	3
1	12

Table 31

Virtual Address Size	Page Size	Page Table Entry Size
32 bits	8 KiB	4 bytes

Table 32

Virtual Address (bits)	Physical DRAM Installed	Page Size	PTE Size (byte)
43	16 GiB	4 KiB	4

Table 33

Entry-ID	Valid	VA Page	Modified	Protection	PA Page
1	1	140	1	RW	30
2	0	40	0	RX	34
3	1	200	1	RO	32
4	1	280	0	RW	31

Table 34

Address of Memory Block Accessed	Hit or Miss	Evicted Block	Contents of Cache Blocks After Reference			
			Set 0	Set 0	Set 1	Set 1
0	Miss		Mem[0]			
1	Miss		Mem[0]		Mem[1]	
2	Miss		Mem[0]	Mem[2]	Mem[1]	
3	Miss		Mem[0]	Mem[2]	Mem[1]	Mem[3]
4	Miss	0	Mem[4]	Mem[2]	Mem[1]	Mem[3]
...						

Table 35

TLB Misses per 1000 Instructions	NPT TLB Miss Latency	Page Faults per 1000 Instructions	Shadowing Page Fault Overhead
0.2	200 cycles	0.001	30,000 cycles

Table 36

Base CPI	Privileged O/S Accesses per 10,000 Instructions	Performance Impact to Trap to the Guest O/S	Performance Impact to Trap to VMM	I/O Access per 10,000 Instructions	I/O Access Time (Includes Time to Trap to Guest O/S)
1.5	120	15 cycles	175 cycles	30	1100 cycles

Table 37

P1	P2
X[0] ++; X[1] = 3;	X[0] = 5; X[1] +=2;

Table 38

P1	P2
A = 1; B = 2; A+=2; B++;	C = B; D = A;

Table 39

	Private	Shared
Benchmark A misses-per-instruction	0.30%	0.12%
Benchmark B misses-per-instruction	0.06%	0.03%

Table 40

Private Cache	Shared Cache	Memory
5	20	180

Table 41

a.	Mesa / gcc
b.	mcf / swim

Table 42

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