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 a local disk or flash memory 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 DDR4. 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 DDR4) or by specifying a block transfer and the starting address. Each command, as well as block transfers, is synchronized with a clock.

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. Price source is https://jcmit.net/memoryprice.htm

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.

Figure 5.7 The cache just before and just after a reference to a word Xn that is not initially in the cache. This reference causes a miss that forces the cache to fetch Xn 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 modulo 8. That is, the low-order log2(8) = 3 bits are used as the cache index. Thus, addresses 00001two, 01001two, 10001two, and 11001two all map to entry 001two of the cache, while addresses 00101two, 01101two, 10101two, and 11101two all map to entry 101two of the cache.

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

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.)

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.

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 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

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.

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 5.22, elements of A and B are read repeatedly to calculate new elements of C. The variables i, j, and k are shown along the rows or columns used to access the arrays.

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 2 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=3. Note that, in contrast to Figure 5.20, a smaller number of elements is accessed.

Figure 5.23 Parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits.

Figure 5.24 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 5.25 Mapping from a virtual to a physical address. The page size is 212=4 KiB. The number of physical pages allowed in memory is 218, since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GiB (230), while the virtual address space is 4 GiB (230).

Figure 5.26 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 212 bytes, or 4 KiB. The virtual address space is 232 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 220, or 1 million entries. The valid bit for each entry indicates whether the mapping is legitimate. 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 5.27 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 5.28 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 5.29 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 5.30 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 426.) 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 5.30 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.

Figure 5.31 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.

Figure 5.32 MIPS control registers. These are considered to be in coprocessor 0, and hence are read using mfc0 and written using mtc0.

Figure 5.33 MIPS code to save and restore state on an exception.

Figure 5.34 The key quantitative design parameters that characterize the major elements of memory hierarchy in a computer. These are typical values for these levels as of 2020. Although the range of values is wide, this is partially because many of the values that have shifted over time are related; for example, as caches become larger to overcome larger miss penalties, block sizes also grow. While not shown, server microprocessors today also have L3 caches, which can be 4 to 50 MiB and contain many more blocks than L2 caches. L3 caches lower the L2 miss penalty to 30 to 40 clock cycles.

Figure 5.35 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 5.36 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.

Figure 5.37 Memory hierarchy design challenges.

Figure 5.38 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 5.39 Four states of the simple controller.

Figure 5.40 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!

Figure 5.41 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 optimization 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.

Figure 5.42 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.

Figure 5.43 Caches in the ARM Cortex-A53 and Intel Core i7 6700. The miss penalty on the A53 is 13 clock cycles for the L1 cache and 124 for the L2 cache.

Figure 5.44 The data miss rate for ARM with a 32 KiB L1 and the global data miss rate for a 1 MiB L2 using the SPECInt2006 benchmarks are significantly affected by the applications. 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. The mcf benchmark is known as a cache buster.

Figure 5.45 The average memory access penalty per data memory reference coming from L1 and L2 is shown for the A53 processor when running SPECInt2006. 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 5.46 The L1 data cache miss rate for the SPECint2006 benchmarks is shown in two ways relative to the demand L1 reads for demand accesses (excluding prefetched). These data, like the rest in this section, were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University (see Peng et al., [2008]).

Figure 5.47 The L2 miss rate shown relative to L1 references to L1.

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 multiple versions of DGEMM as change matrix size measured in billion floating point operations per second (GFLOPS/second). The fully optimized code is 14–32 times as fast as the C version in Chapter 2. Python runs at 0.007 GFLOPS/second for all matrix sizes. The Intel i7 hardware speculates by prefetching from the L3 cache to the L1 and L2 caches, which is why the benefits of blocking are not as high as on some microprocessors.

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]).

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.