Chapter 1

Memory Systems

Yoongu Kim

Carnegie Mellon University

Onur Mutlu

Carnegie Mellon University

1.1 Introduction

As shown in Figure 1.1, a computing system consists of three fundamental units: *(i)* units of computation to perform operations on data (e.g., processors, as we have seen in a previous chapter), *(ii)* units of storage (or memory) that store data to be operated on or archived, *(iii)* units of communication that communicate data between computation units and storage units. The storage/memory units are usually categorized into two: *(i) memory system*, which acts as a *working storage area*, storing the data that is currently being operated on by the running programs, and *(ii)* the backup *storage system*, e.g., the hard disk, which acts as a *backing store*, storing data for a longer term in a persistent manner. This chapter will focus on the “working storage area” of the processor, i.e., the memory system.

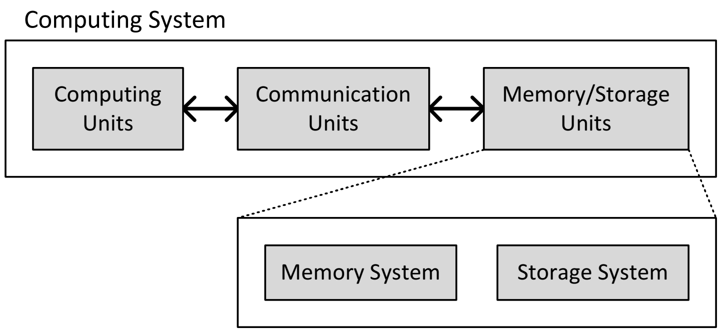


FIGURE 1.1: Computing system

The memory system is the repository of data from where data can be retrieved and updated by the processor (or processors). Throughout the operation of a computing system, the processor reads data from the memory system, performs computation on the data, and writes the modified data back into the memory system – continuously repeating this procedure until all the necessary computation has been performed on all the necessary data.

1.1.1 Basic Concepts and Metrics

The *capacity* of a memory system is the total amount of data that it can store. Every piece of data stored in the memory system is associated with a unique *address*. For example, the first piece of data has an address of 0, whereas the last piece of data has an address of $capacity - 1$. The full range of possible addresses, spanning from 0 to $capacity - 1$, is referred to as the *address space* of the memory system. Therefore, in order to access a particular piece of data from the memory system, the processor must supply its address to the memory system.

The *performance* of a memory system is characterized by several important metrics: *(i)* latency, *(ii)* bandwidth, and *(iii)* parallelism. A high-performance memory system would have low latency, high bandwidth, and high parallelism. *Latency* is the amount of time it takes for the processor to access one piece of data from the memory system. *Bandwidth*, also known as *throughput*, is the rate at which the processor can access pieces of data from the memory system. At first blush, latency and bandwidth appear to be inverses of each other. For example, if it takes time T seconds to access one piece of data, then it would be tempting to assume that $\frac{1}{T}$ pieces of data can be accessed over the duration of 1 second. However, this is not always true. To fully understand the relationship between latency and bandwidth, we must also examine the third metric of a memory system's performance. *Parallelism* is the number of accesses to the memory system that can be served concurrently. If a memory system has a parallelism of 1, then all accesses are served one-at-a-time, and this is the only case in which bandwidth is the inverse of latency. But, if a memory system has a parallelism of more than 1, then multiple accesses to different addresses can be served concurrently, thereby overlapping their latencies. For example, when the parallelism is equal to two, during the amount of time required to serve one access (T), another access is served in parallel. Therefore, while the latency of an individual access remains unchanged at T, the bandwidth of the memory system doubles to $\frac{2}{T}$. More generally, the relationship among latency, bandwidth, and parallelism of a memory system can be expressed as follows.

$$Bandwidth\ [accesses/time] = \frac{Parallelism\ [unitless]}{Latency\ [time/access]}$$

While the bandwidth can be measured in the unit of accesses-per-time (equation above), another way of expressing bandwidth is in the unit of bytes-per-time – i.e., the-amount-of-data-accessed-per-time (equation below).

$$Bandwidth\ [bytes/time] = \frac{Parallelism\ [unitless]}{Latency\ [time/access]} \times DataSize\ [bytes/access]$$

An additional characteristic of a memory system is cost. The *cost* of a memory system is the capital expenditure required to implement it. Cost is closely related to the capacity and performance of the memory system: increasing the capacity and performance of a memory system *usually* also makes it more expensive.

1.1.2 Two Components of the Memory System

When designing a memory system, computer architects strive to optimize for all of the three characteristics explained above: large capacity, high performance, and low cost. However, a memory system that has both large capacity and high performance is also very expensive. For example, when one wants to increase the capacity of memory, it is almost always the case that the latency of memory also increases. Therefore, when designing a memory system within a reasonable cost budget, there exists a fundamental trade-off relationship between capacity and performance: it is possible to achieve either large capacity or high performance, but not both at the same time in a cost-effective manner.

As a result of the trade-off between capacity and performance, a modern memory system typically consists of two components, as shown in Figure 1.2: *(i)* a *cache* [44] (pronounced as "cash"), a component that is small but relatively fast-to-access and *(ii) main memory*, a component that is large but relatively slow-to-access. Between the two, main memory is the all-encompassing (i.e., “main”) repository whose capacity is usually determined to minimize accesses to the storage system. However, if the memory system were to consist only of main memory, then all accesses to the memory system would be served by main memory, which is slow. That is why a memory system also contains a cache: although a cache is much smaller than main memory, it is fast (exactly because it is smaller). The memory system utilizes the cache by taking a subset of the data from main memory and placing it into the cache – thereby enabling some of the accesses to the memory system to be served quickly by the cache. The data stored in the cache are replicas of the data that are already already stored in main memory. Hence, the capacity of the memory system as a whole is defined by the capacity of only the main memory while ignoring the capacity of the cache. In other words, while the processor can always access the data it wants from main memory, the cache exists to *expedite* some of those accesses as long as they are to data that are replicated in the cache. In fact, most memory systems employ not just one, but *multiple* caches, each of which provides a different trade-off between capacity and performance. For example, there can be two caches, one of which is even smaller and faster than the other. In this case, the data in the smaller cache is a subset of the data in the larger cache, similar to how the larger cache is a subset of the main memory.

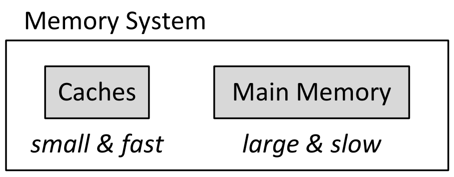


FIGURE 1.2: Memory system

Note that the structure and operation of the hardware components that make up the cache and main memory can be similar (in fact, they can be exactly the same). However, the structure and operation of cache and memory components are affected by *(i)* the function of the respective components and *(ii)* the technology in which they are implemented. The main function of caches is to store a small amount of data such that it can be accessed quickly. Traditionally, caches have been designed using the SRAM technology (so that they are fast), and main memory has been designed using the DRAM technology (so that it has large capacity). As a result, caches and main memory have evolved to be different in structure and operation, as we describe in later sections (Section 1.4 and Section 1.5).

1.2 Memory Hierarchy

An ideal memory system would have the following three properties: high performance (i.e., low latency and high bandwidth), large capacity, and low cost. Realistically, however, technological and physical constraints limit the design of a memory system that achieves all three properties at the same time. Rather, one property can be improved only at the expense of another – i.e., there exists a fundamental *trade-off* relationship that tightly binds together performance, capacity, and cost of a memory system. (To simplify the discussion, we will assume that we always want low cost, restricting ourselves to the "zero-sum" trade-off relationship between performance and capacity.) For example, main memory is a component of the memory system that provides large capacity but at relatively low performance, while a cache is another component of the memory system that provides high performance but at relatively small capacity. Therefore, caches and main memory lie at opposite ends of the trade-off spectrum between performance and capacity.

We have mentioned earlier that modern memory systems consist of both caches and main memory. The reasoning behind this is to achieve the best of both worlds (performance and capacity) – i.e., a memory system that has the high performance of a cache and the large capacity of main memory. If the memory system consists only of main memory, then *every* access to the memory system would experience the high latency (i.e., low performance) of main memory. However, if caches are used in addition to main memory, then *some* of the accesses to the memory system would be served by the caches at low latency, while the remainder of the accesses are assured to find their data in main memory due to its large capacity – albeit at high latency. Therefore, as a net result of using both caches and main memory, the memory system's *effective latency* (i.e., average latency) becomes lower than that of main memory, while still retaining the large capacity of main memory.

Within a memory system, caches and main memory are said to be part of a *memory hierarchy*. More formally, a memory hierarchy refers to how multiple components (e.g., cache and main memory) with different performance/capacity properties are combined together to form the memory system. As shown in Figure 1.3, at the "top-level" of the memory hierarchy lies the fastest (but the smallest) component, whereas at the "bottom-level" of the memory hierarchy lies the slowest (but the largest) component. Going from top to bottom, the memory hierarchy typically consists of multiple levels of caches (e.g., L1, L2, L3 caches in Figure 1.3, standing for respectively level-1, level-2, level-3) – each of whose capacity is larger than the one above it – and a single level of main memory at the bottom whose capacity is the largest. The bottom-most cache (e.g., the L3 cache in Figure 1.3 which lies immediately above main memory) is also referred to as the *last-level cache*.

When the processor accesses the memory system, it typically does so in

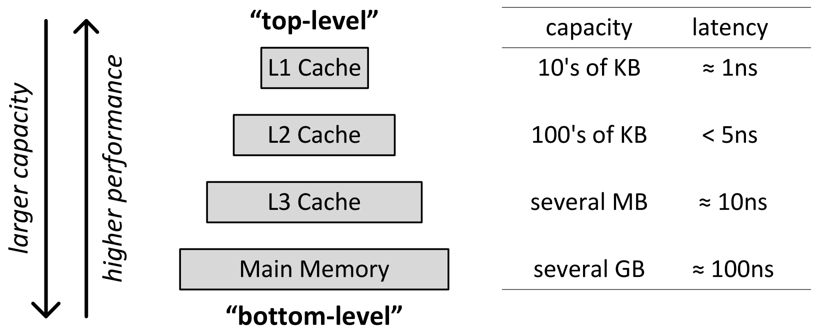


FIGURE 1.3: Example of a memory hierarchy with three levels of caches

a sequential fashion by first searching the top component to see whether it contains the necessary data. If the data is found, then the access is said to have "hit" in the top component and the access is served there on-the-spot without having to search the lower components of the memory hierarchy. Otherwise, the access is said to have "missed" in the top component and it is passed down to the immediately lower component in the memory hierarchy, where the access may again experience either a hit or a miss. As an access goes lower into the hierarchy, the probability of a hit becomes greater due to the increasing capacity of the lower components, until it reaches the bottom-level of the hierarchy where it is guaranteed to always hit (assuming the data is in main memory).

Assuming a two-level memory hierarchy with a single cache and main memory, the effective latency of the memory system can be expressed by the following equation. In the equation, P^{hit}_{cache} denotes the probability that an access would hit in the cache, also know as the cache *hit-rate*.

$$Latency_{effective} = P^{hit}_{cache} \times Latency_{cache} + (1 - P^{hit}_{cache}) \times Latency_{main\_memory}$$

- $0 \leq P^{hit}_{cache} \leq 1$
- $Latency_{cache} \ll Latency_{main\_memory}$

Similarly, for a three-level memory hierarchy with two caches and main memory, the effective latency of the memory system can be expressed by the following equation.

$$Latency_{effective} = P^{hit}_{cache1} \times Latency_{cache1} +$$
$$(1 - P^{hit}_{cache1}) \times \{P^{hit}_{cache2} \times Latency_{cache2} + (1 - P^{hit}_{cache2}) \times Latency_{main\_memory}\}$$

- $0 \leq P^{hit}_{cache1}, P^{hit}_{cache1} \leq 1$
- $Latency_{cache1} < Latency_{cache2} \ll Latency_{main\_memory}$

As both equations show, a high hit-rate in the cache implies a lower effective latency. In the best case, when all accesses hit in the cache ($P^{hit}_{cache} = 1$), then the memory hierarchy has the lowest effective latency, equal to that of the cache. While having a high hit-rate is always desirable, the actual value of the hit-rate is determined primarily by *(i)* the cache's size and *(ii)* the processor's memory access behavior. First, compared to a small cache, a large cache is able to store more data and has a better chance that a given access will hit in the cache. Second, if the processor tends to access a small set of data over and over again, the cache can store those data so that subsequent accesses will hit in the cache. In this case, a small cache would be sufficient to achieve a high hit-rate.

Fortunately, many computer programs – that the processor executes – access the memory system in this manner. In other words, many computer programs exhibit *locality* in their memory access behavior. Locality exists in two forms: *temporal locality* and *spatial locality.* First, given a piece of data that has been accessed, temporal locality refers to the phenomenon (or memory access behavior) in which the same piece of data is likely to be accessed again in the near future. Second, given a piece of data that has been accessed, spatial locality refers to the phenomenon (or memory access behavior) in which neighboring pieces of data (i.e., data at nearby addresses) are likely to be accessed in the near future. Thanks to both temporal/spatial locality, the cache – and, more generally, the memory hierarchy – is able to reduce the effective latency of the memory system.

1.3 Managing the Memory Hierarchy

As mentioned earlier, every piece of data within a memory system is associated with a unique address. The set of all possible addresses for a memory system is called the address space and it ranges from 0 to $capacity - 1$, where *capacity* is the size of the memory system. Naturally, when software programmers compose computer programs, they rely on the memory system and assume that specific pieces of the program's data can be stored at specific addresses without any restriction, as long as the addresses are valid – i.e., the addresses do not overflow the address space provided by the memory system. In practice, however, the memory system does *not* expose its address space directly to computer programs. This is due to two reasons.

First, when a computer program is being composed, the modern software programmer has no way of knowing the capacity of the memory system on which the program will run. For example, the program may run on a computer that has a very small large memory system (e.g., 1 TB capacity) or a very small memory system (e.g., 1 MB capacity). While an address of 1 GB is perfectly valid for the first memory system, the same address is invalid for the second memory system, since it exceeds the maximum bound (1 MB) of the address space. As a result, if the address space of the memory system is directly exposed to the program, the software programmer can never be sure which addresses are valid and can be used to store data for the program she is composing.

Second, when a computer program is being composed, the software programmer has no way of knowing which other programs will run simultaneously with the program. For example, when the user runs the program, it may run on the same computer as many other different programs, all of which may happen to utilize the same address (e.g., address 0) to store a particular piece of their data. In this case, when one program modifies the data at that address, it overwrites another program's data that was stored at the same address, even though it should not be allowed to. As a result, if the address space of the memory system is directly exposed to the program, then multiple programs may overwrite and corrupt each other's data, leading to incorrect execution for all of the programs.

1.3.1 Virtual vs. Physical Address Spaces

As a solution to these two problems, a memory system does not directly expose its address space, but instead provides the illusion of an extremely large address space that is separate for each individual program. While the illusion of a *large* address space solves the first problem of different capacities across different memory systems, the illusion of a *separate* large address space for each individual program solves the second problem of multiple programs modifying data at the same address. Since such large and separate address spaces are only an illusion provided to the programmer to make her life easier in composing programs, they are referred to as *virtual address spaces*. In contrast, the actual underlying address space of the memory system is called the *physical address space*. To use concrete numbers from today's systems (circa 2013), the physical address space of a typical 8 GB memory system ranges from 0 to $8\ GB - 1$, whereas the virtual address space for a program typically ranges from 0 to $256\ TB - 1$.[1]

A virtual address, just by itself, does not represent any real storage location unless it is actually backed up by a physical address. It is the job of the operating system – a program that manages all other programs as well as resources in the computing system – to substantiate a virtual address by mapping it to a physical address. For example, at the very beginning of when a program is executed, none of its virtual addresses are mapped to physical addresses. At this point, if the program attempts to store a piece of data to a particular virtual address, the operating system must first intervene on behalf

[1]For example, the current generation of mainstream x86-64 processors manufactured by Intel uses 48-bit virtual addresses – i.e., a 256 TB virtual address space ($2^{48} = 256\ T$) [13].

of the program: the operating system maps the virtual address to a physical address that is *free* – i.e., a physical address that is not yet mapped to another virtual address. Once this mapping has been established, it is memorized by the operating system and used later for "translating" any subsequent access to that virtual addresses to its corresponding physical address (where the data is stored).

1.3.2 Virtual Memory System

The entire mechanism, described so far, in which the operating system maps and translates between virtual and physical addresses is called *virtual memory.* There are three considerations when designing a virtual memory system as part of an operating system: *(i)* when to map a virtual address to a physical address, *(ii)* the mapping granularity, and *(iii)* what to do when physical addresses are exhausted.

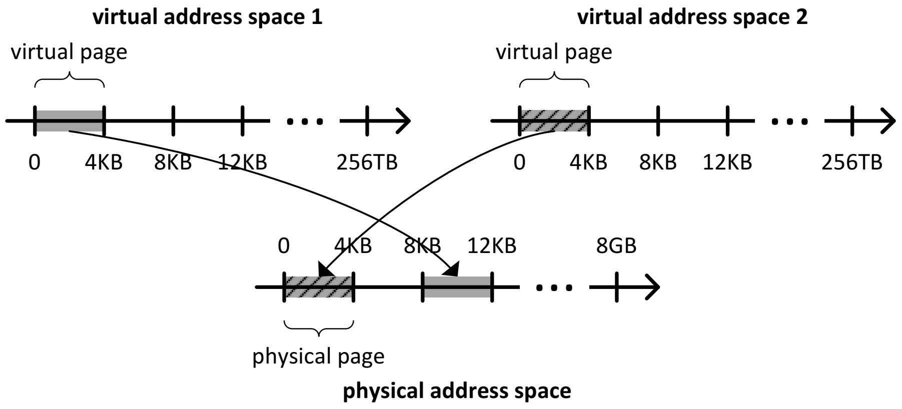


FIGURE 1.4: Virtual memory system

First, most virtual memory systems map a virtual address when it is accessed for the very first time – i.e., *on-demand*. In other words, if a virtual address is never accessed, it is never mapped to a physical address. Although the virtual address space is extremely large (e.g., 256 *TB*), in practice, only a very small fraction of it is actually utilized by most programs. Therefore, mapping the entirety of virtual address space to the physical address space is wasteful, because the overwhelming majority of the virtual addresses will never be accessed. Not to mention the fact that the virtual address space is much larger than the physical address space such that it is not possible to map all virtual addresses to begin with.

Second, a virtual memory system must adopt a granularity at which it maps addresses from the virtual address space to the physical address space. For example, if the granularity is set equal to 1 byte, then the virtual memory system evenly divides the virtual/physical address into 1-byte virtual/physical "chunks," respectively. Then, the virtual memory system can arbitrarily map a 1-byte virtual chunk to any 1-byte physical chunk, as long as the physical chunk is free. However, such a fine division of the address spaces into large numbers of small chunks has a major disadvantage: it increases the complexity of the virtual memory system. As we recall, once a mapping between a pair of virtual/physical chunks is established, it must be memorized by the virtual memory system. Hence, large numbers of virtual/physical chunks imply a large number of possible mappings between the two, which increases the bookkeeping overhead of memorizing the mappings. To reduce such an overhead, most virtual memory systems coarsely divide the address spaces into smaller numbers of chunks, where a chunk is called a *page* and whose typical size is 4 KB. As shown in Figure 1.4, a 4 KB chunk of the virtual address space is referred to as a *virtual page*, whereas a 4 KB chunk of the physical address is referred to as a *physical page* (alternatively, a *frame*). Every time a virtual page is mapped to a physical page, the operating system keeps track of the mapping by storing it in a data structure called the *page table*.

Third, as a program accesses a new virtual page for the very first time, the virtual memory system maps the virtual page to a free physical page. However, if this happens over and over, the physical address space may become exhausted – i.e., none of the physical pages are free since all of them have been mapped to virtual pages. At this point, the virtual memory system must "create" a free physical page by reclaiming one of the mapped physical pages. The virtual memory system does so by *evicting* a physical page's data from main memory and "un-mapping" the physical page from its virtual page. Once a free physical page is created in such a manner, the virtual memory system can map it to a new virtual page. More specifically, the virtual memory system takes the following three steps in order to reclaim a physical page and map it to a new virtual page. First, the virtual memory system selects the physical page that will be reclaimed, i.e., the *victim.* The selection process of determining the victim is referred to as the *page replacement policy* [5]. While the simplest policy is randomly selecting any physical page, such a policy may significantly degrade the performance of the computing system. For example, if a very frequently accessed physical page is selected as the victim, then a future access to that physical page would be served by the hard disk. However, since a hard disk is extremely slow compared to main memory, the access would incur a very large latency. Instead, virtual memory systems employ more sophisticated page replacement policies that try to select a physical page that is unlikely to be accessed in the near future, in order to minimize the performance degradation. Second, after a physical page has been selected as the victim, the virtual memory system decides whether the page's data should be migrated out of main memory and into the hard disk. If the page's data had been modified by the program while it was in main memory, then the page must be written back into the hard disk – otherwise, the modifications that were made to the page's data would be lost. On the other hand, if the page's data had *not* been modified, then the page can simply be evicted from main memory (without being written into the hard disk) since the program can always retrieve the page's original data from the hard disk. Third, the operating system updates the page table so that the virtual page (that had previously mapped to the victim) is now mapped to the hard disk instead of a physical page in main memory. Finally, now that a physical page had been reclaimed, the virtual memory system maps the free physical page to a new virtual page and updates the page table accordingly.

After the victim has been evicted from main memory, it would be best if the program does not access the victim's data ever again. This is because accessing the victim's data incurs the large latency of the hard disk where it is stored. However, if the victim is eventually accessed, then the virtual memory system brings the victim's data back from the hard disk and places it into a free physical page in main memory. Unfortunately, if main memory has no free physical pages remaining at this point, then another physical page must be chosen as a victim and be evicted from main memory. If this happens repeatedly, different physical pages are forced to *ping-pong* back and forth between main memory and hard disk. This phenomenon, referred to as *swapping* or *thrashing*, typically occurs when the capacity of the main memory is not large enough to accommodate all of the data that a program is actively accessing (i.e., its working set). When a computing system experiences swapping, its performance degrades detrimentally since it must constantly access the extremely slow hard disk instead of the faster main memory.

1.4 Caches

Generally, a cache is any structure that stores data that is likely to be accessed again (e.g., frequently accessed data or recently accessed data) in order to avoid the long latency operation required to access the data from a much slower structure. For example, web servers on the internet typically employ caches that store the most popular photographs or news articles so that they can be retrieved quickly and sent to the end user. In the context of the memory system, a cache refers to a small but fast component of the memory hierarchy that stores the most recently (or most frequently) accessed data among all data in the memory system [44, 26]. Since a cache is designed to be faster than main memory, data stored in the cache can be accessed quickly by the processor. The effectiveness of a cache depends on whether a large fraction of the memory accesses "hits" in the cache and, as a result, are able to avoid being served by the much slower main memory. Despite its small capacity, a cache can still achieve a high hit-rate thanks to the fact that many computer programs exhibit locality (Section 1.2) in their memory access behavior: data that have been accessed in the past are likely to be accessed again in the future. That is why a small cache, whose capacity is much less than that of main memory, is able to serve most of the memory accesses as long as the cache stores the most recently (or most frequently) accessed data.

1.4.1 Basic Design Considerations

There are three basic design considerations when implementing a cache: *(i)* physical substrate, *(ii)* capacity, and *(iii)* granularity of managing data. Below, we discuss each of them in that order.

First, the physical substrate used to implement the cache must be able to deliver much lower latencies than that used to implement main memory. That is why *SRAM* (static random-access-memory, pronounced "es-ram") has been – and continues to be – by far the most dominant physical substrate for caches. The primary advantage of SRAM is that it can operate at very high speeds that are on par with the processor itself. In addition, SRAM consists of the same type of semiconductor-based transistors that make up the processor. (This is not the case with DRAM, which is used to implement main memory, as we will see in Section 1.5.) So an SRAM-based cache can be placed – at low cost – side-by-side with the processor on the same semiconductor chip, allowing it to achieve even lower latencies. The smallest unit of SRAM is an *SRAM cell* which typically consists of six transistors. The six transistors collectively store a single bit of data (0 or 1) in the form of electrical voltage ("low" or "high"). When the processor reads from an SRAM cell, the transistors feed the processor with the data value that corresponds to their voltage levels. On the other hand, when the processor writes into an SRAM cell, the voltage of the transistors are appropriately adjusted to reflect the updated data value that is being written.

Second, when determining the capacity of a cache, one must be careful to balance the need for high hit-rate, low cost, and low latency. While a large cache is more likely to provide a high hit-rate, it also has two shortcomings. First, a large cache has high cost due to the increased number of transistors that are required to implement it. Second, a large cache is likely to have a higher latency. This is because orchestrating the operation of many transistors is a complex task that introduces extra overhead delays (e.g., it may take a longer time to determine the location of an address). In practice, a cache is made large enough to achieve a sufficiently high hit-rate, but not large enough to incur significantly high cost and latency.

Third, a cache is divided into many small pieces, called *cache blocks*, as shown in Figure 1.5. A cache block is the granularity in which the cache manages data. The typical size of a cache block is 64 bytes in modern memory systems. For example, a 64 KB cache consists of 1024 separate cache blocks. Each of these cache blocks can store data that corresponds to an arbitrary 64 byte "chunk" of the address space – i.e., any given 64-byte cache block can contain data from address 0, or address 64, or address 128, address 192, and so on. Therefore, a cache block requires some sort of a "label" that conveys the address of the data stored in the cache block. For exactly this purpose, every cache block has its own *tag* where the address of the data (not the data itself) is stored. When the processor accesses the cache for a piece of data at a particular address, it searches the cache for the cache block whose tag matches the address. If such a cache block exists, then the processor accesses the data contained in the cache block – as explained earlier, this is called a cache hit. In addition to its address, a cache block's tag may also store other types of information about the cache block. For example, whether the cache block is empty, whether the cache block has been written to, or how recently the cache block has been accessed. These topics and more will soon be discussed in this section.

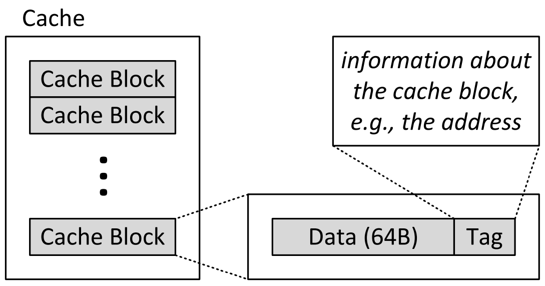


FIGURE 1.5: Cache

1.4.2 Logical Organization

In addition to the basic considerations discussed above, the computer architect must decide how a cache is logically organized – i.e., how it maps 64 byte chunks of the address space onto its 64 byte cache blocks. Since the memory system's address space is much larger than the cache's capacity, there are many more chunks than there are cache blocks. As a result, the mapping between chunks and cache blocks is necessarily "many-to-few." However, for each individual chunk of the address space, the computer architect can design a cache that has the ability to map the chunk to *(i)* any of its cache blocks or *(ii)* only a specific cache block. These two different logical organizations represent two opposite extremes in the degree of freedom when mapping a chunk to a cache block. This freedom, in fact, presents a crucial trade-off between the complexity and utilization of the cache, as described below.

On the one hand, a cache that provides the greatest freedom in mapping a chunk to a cache block is said to have a *fully-associative* organization (Figure 1.6, lower-left). When a new chunk is brought in from main memory, a fully-associative cache does not impose any restriction on where the chunk can be placed – i.e., the chunk can be stored in *any* of the cache blocks. Therefore, as long as the cache has at least one empty cache block, the chunk is guaranteed to be stored in the cache without having to make room for it by evicting an already occupied (i.e., non-empty) cache block. In this regard, a fully-associative cache is the best at efficiently utilizing all the cache blocks in the cache. However, the downside of a fully-associative cache is that the processor must exhaustively search *all* cache blocks whenever it accesses the cache, since any one of the cache blocks may contain the data that the processor wants. Unfortunately, searching through all cache blocks not only takes a long time (leading to high access latency), but also wastes energy.

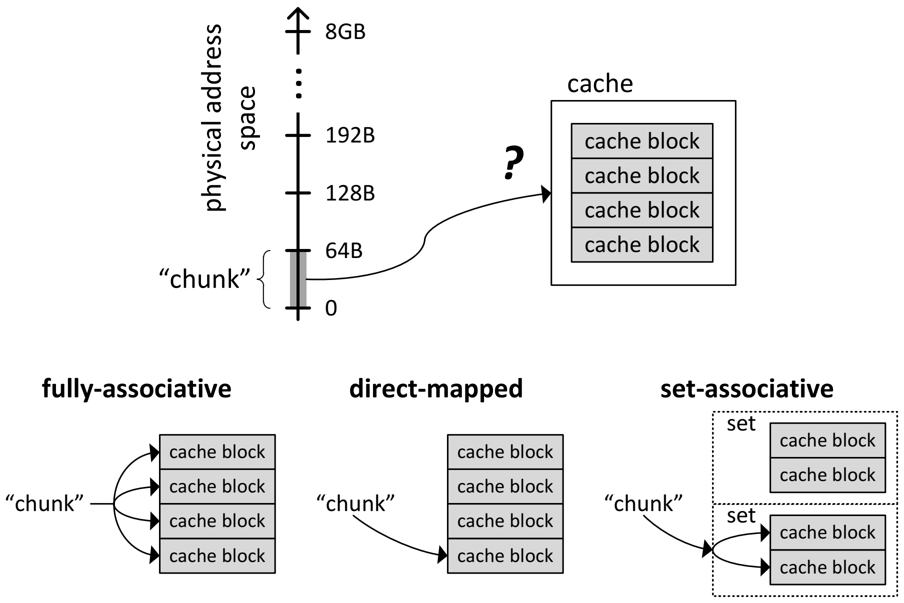


FIGURE 1.6: Logical organization

On the other hand, a cache that provides the least freedom in mapping a chunk to a cache block is said to have a *direct-mapped* organization (Figure 1.6, lower-middle). When a new chunk is brought in from main memory, a direct-mapped cache allows the chunk to be placed in only *a specific* cache block. For example, let us assume a 64 KB cache consisting of 1024 cache blocks (64 bytes). A simple implementation of a direct-mapped cache would map every 1024th chunk (64 bytes) of the address space to the same cache block – e.g., chunks at address 0, address 64K, address 128K, etc. would all map to the 0th cache block in the cache. But if the cache block is already occupied with a different chunk, then the old chunk must first be evicted before a new chunk can be stored in the cache block. This is referred to as a *conflict* – i.e., when two different chunks (corresponding to two different addresses) contend with each other for the same cache block. In a direct-mapped cache, conflicts can occur at one cache block even when all other cache blocks are empty. In this regard, a direct-mapped cache is the worst at efficiently utilizing all the cache blocks in the cache. However, the upside of a direct-mapped cache is that the processor can simply search only *one* cache block to quickly determine whether the cache contains the data it wants. Hence, the access latency of a direct-mapped cache is low.

As a middle ground between the two organizations (fully-associative vs. direct-mapped), there is a third alternative called the *set-associative* organization [12] (Figure 1.6, lower-right), which allows a chunk to map to one of multiple (but not all) cache blocks within a cache. If a cache has a total of N cache blocks, then a fully-associative organization would map a chunk to any of the N cache blocks, while a direct-mapped organization would map a chunk to only 1 specific cache block. A set-associative organization, in contrast, is based on the concept of *sets*, which are small non-overlapping groups of cache blocks. A set-associative cache is similar to a direct-mapped cache in that a chunk is mapped to only one specific set. However, a set-associative cache is also similar to a fully-associative cache in that the chunk can map to any cache block that belongs to the specific set. For example, let us assume a set-associative cache in which each set consists of 2 cache blocks, which is called a *2-way* set-associative cache. Initially, such a cache maps a chunk to one specific set out of all $\frac{N}{2}$ sets. Then, within the set, the chunk can map to either of the 2 cache blocks that belong to the set. More generally, a W-way set-associative cache ($1 < W < N$) directly maps a chunk to one specific set, while fully-associatively mapping a chunk to any of the W cache block within the set. For a set-associative cache, the value of W is fixed when the cache is designed and cannot be changed afterwards. However, depending on the value of W, a set-associative cache can behave similarly to a fully-associative cache (for large values of W) or a direct-mapped cache (for small values of W). In fact, an N-way set-associative cache degenerates into a fully-associative cache, whereas a 1-way set-associative cache degenerates into a direct-mapped cache.

1.4.3 Management Policies

If a cache has unlimited capacity, it would have the highest hit-rate since it can store all of the processor's data that is in memory. Realistically, however, a cache has only a limited capacity. Therefore, it needs to be selective about which data it should store – i.e., the cache should store only the data that is the most likely to be accessed by the processor in the near future. In order to achieve that goal, a cache employs various management policies that enable it to make the best use of its small capacity: *(i)* allocation policy and *(ii)* replacement policy. Below, we discuss each of them respectively.

First, the cache's *allocation policy* determines how its cache blocks become populated with data. Initially, before the execution of any program, all of the cache blocks are empty. As a program starts to execute, it accesses new chunks from the memory system that are not yet stored in the cache. Whenever this happens (i.e., a cache miss), the cache's allocation policy decides whether the new chunk should be stored in one of the empty cache blocks. Since most programs exhibit locality in their memory accesses, a chunk that is accessed (even for the first time) is likely to be accessed again in the future. That is why *always-allocate* is one of the most popular allocation policies: for every cache miss, the always-allocate policy populates an empty cache block with the new chunk. (On the other hand, a different allocation policy may be more discriminative and prevent certain chunks from being allocated in the cache.) However, when the cache has no empty cache blocks left, a new chunk cannot be stored in the cache unless the cache "creates" an empty cache block by reclaiming one of the occupied cache blocks. The cache does so by *evicting* the data stored in an occupied cache block and *replacing* it with the new chunk, as described next.

Second, when the cache does not have an empty cache block where it can store a new chunk, the cache's *replacement policy* selects one of the occupied cache blocks to evict, i.e., the *victim* cache block. The replacement policy is invoked when a new chunk is brought into the cache. Depending on the cache's logical organization, the chunk may map to one or more cache blocks. However, if all such cache blocks are already occupied, the replacement policy must select one of the occupied cache blocks to evict from the cache. For a direct-mapped cache, the replacement policy is trivial: since a chunk can be mapped to only one specific cache block, then there is no choice but to evict that specific cache block if it is occupied. Therefore, a replacement policy applies only to set-associative or fully-associative caches, where a chunk can potentially be mapped to one of multiple cache blocks – any one of which may become the victim if all of those cache blocks are occupied. Ideally, the replacement policy should select the cache block that is expected to be accessed the farthest away in the future, such that evicting the cache block has the least impact on the cache's hit-rate [3]. That is why one of the most common replacement policy is the *LRU* (*least-recently-used*) policy, in which the cache block that has been the least recently accessed is selected as the victim. Due to the principle of locality, such a cache block is less likely to be accessed in the future. Under the LRU policy, the victim is the least recently accessed cache block among *(i)* all cache blocks within a set (for a set-associative cache) or *(ii)* among all cache blocks within the cache (for a fully-associative cache). To implement the LRU policy, the cache must keep track of each block in terms of the last time when it was accessed. A similar replacement policy is the *LFU* (*least-frequently-used*) policy, in which the cache block that has been the least frequently accessed is selected as the victim. We refer the reader to the following works for more details on cache replacement policies: Liptay [26] and Qureshi et al. [36, 39].

1.4.4 Managing Multiple Caches

Until now, we have discussed the design issues and management policies for a single cache. However, as described previously, a memory hierarchy typically consists of more than just one cache. In the following, we discuss the policies that govern how multiple caches within the memory hierarchy interact with each other: *(i)* inclusion policy, *(ii)* write handling policy, and *(ii)* partitioning policy.

First, the memory hierarchy may consist of multiple levels of caches in addition to main memory. As the lowest level, main memory always contains the superset of all data stored in any of the caches. In other words, main memory is *inclusive* of the caches. However, the same relationship may not hold between one cache and another cache depending on the *inclusion policy* employed by the memory system [2]. There are three different inclusion policies: *(i)* inclusive, *(ii)* exclusive, and *(iii)* non-inclusive. First, in the *inclusive* policy, a piece of data in one cache is guaranteed to be also found in all higher levels of caches. Second, in the *exclusive* policy, a piece of data in one cache is guaranteed *not* to be found in all higher levels of caches. Third, in the *non-inclusive* policy, a piece of data in one cache may or may not be found in higher levels of caches. Among the three policies, the inclusive and exclusive policies are opposites of each other, while all other policies between the two are categorized as non-inclusive. On the one hand, the advantage of the exclusive policy is that it does not waste cache capacity since it does not store multiple copies of the same data in all of the caches. On the other hand, the advantage of the inclusive policy is that it is simplifies searching for data when there are multiple processors in the computing system. For example, if one processor wants to know whether another processor has the data it needs, it does not need to search all levels of caches of that other processor, but instead search only the largest cache. (This is related to the concept of *cache coherence* which is not covered in this chapter.) Lastly, the advantage of the non-inclusive policy is that it does not require the effort to maintain a strict inclusive/exclusive relationship between caches. For example, when a piece of data is inserted into one cache, inclusive or exclusive policies require that the same piece of data be inserted into or evicted from other levels of caches. In contrast, the non-inclusive policy does not have this requirement.

Second, when the processor writes new data into a cache block, the data stored in the cache block is modified and becomes different from the data that was originally brought into the cache. While the cache contains the newest copy of the data, all lower levels of the memory hierarchy (i.e., caches and main memory) still contain an old copy of the data. In other words, when a write access hits in a cache, a discrepancy arises between the modified cache block and the lower levels of the memory hierarchy. The memory system resolves this discrepancy by employing a *write handling policy.* There are two types of write handling policies: *(i)* write-through and *(ii)* write-back. First, in a *write-through policy*, every write access that hits in the cache is propagated down to the lowest levels of the memory hierarchy. In other words, when a cache at a particular level is modified, the same modification is made for all lower levels of caches and for main memory. The advantage of the write-through policy is that it prevents any data discrepancy from arising in the first place. But, its disadvantage is that every write access is propagated through the entire memory hierarchy (wasting energy and bandwidth), even when the write access hits in the cache. Second, in a *write-back policy*, a write access that hits in the cache modifies the cache block at only that cache, without being propagated down the rest of the memory hierarchy. In this case, however, the cache block contains the only modified copy of the data, which is different from the copies contained in lower levels of caches and main memory. To signify that the cache block contains modified data, a write-back cache must have a *dirty flag* in the tag of each cache block: when set to '1', the dirty flag denotes that the cache block contains modified data. Later on, when the cache block is evicted from the cache, it must be written into the immediately lower level in the memory hierarchy, where the dirty flag is again set to '1'. Eventually, through a cascading series of evictions at multiple levels of caches, the modified data is propagated all the way down to main memory. The advantage of the write-back policy is that it can prevent write accesses from always being written into all levels of the memory hierarchy – thereby conserving energy and bandwidth. Its disadvantage is that it slightly complicates the cache design since it requires additional dirty flags and special handling when modified cache blocks are evicted.

Third, a cache at one particular level may be partitioned into two smaller caches, each of which is dedicated to two different types of data: *(i)* instruction and *(ii)* data. Instructions are a special type of data that tells the computer how to manipulate (e.g., add, subtract, move) other data. Having two separate caches (i.e., an instruction cache and a data cache) has two advantages. First, it prevents one type of data from monopolizing the cache. While the processor needs both types of data to execute a program, if the cache is filled with only one type of data, the processor may need to access the other type of data from lower levels of the memory hierarchy, thereby incurring a large latency. Second, it allows each of the caches to be placed closer to the processor – lowering the latency to supply instructions and data to the processor. Typically, one part of the processor (i.e., the instruction fetch engine) accesses instructions, while another part of the processor (i.e., the data fetch engine) accesses non-instruction data. In this case, the two caches can each be co-located with the part of the processor that accesses its data – resulting in lower latencies and potentially higher operating frequency for the processor. For this reason, only the highest level of the cache in the memory hierarchy, which is *directly* accessed by the processor, is partitioned into an instruction cache and a data cache.

1.4.5 Specialized Caches for Virtual Memory

Specialized caches have been used to accelerate address translation in virtual memory systems (discussed in Section 1.3). The most commonly used such cache is referred to as a *TLB* (translation lookaside buffer). The role of a TLB is to cache the recently used virtual to physical address translations by the processor such that the translation is quick for the virtual addresses that hit in the TLB. Essentially, a TLB is a cache that caches the parts of the page table that are recently used by the processor.

1.5 Main Memory

While caches are predominantly optimized for high speed, in contrast, the foremost purpose of main memory is to provide as large capacity as possible at low cost and at reasonably low latency. That is why the preferred physical substrate for implementing main memory is *DRAM* (dynamic random-access-memory, pronounced “dee-ram”) [6]. The smallest unit of DRAM is a *DRAM cell*, consisting of one transistor (1T) and one capacitor (1C). A DRAM cell stores a single bit of data (0 or 1) in the form of electrical charge in its capacitor (“discharged” or “charged”). DRAM’s primary advantage over SRAM lies in its small cell size: a DRAM cell requires fewer electrical components (1T and 1C) than an SRAM cell (6T). Therefore, many more DRAM cells can be placed in the same amount of area on a semiconductor chip, enabling DRAM-based main memory to achieve a much larger capacity for approximately the same amount of cost.

Typically, SRAM-based caches are integrated on the same semiconductor chip as the processor. In contrast, DRAM-based main memory is implemented using one or more dedicated DRAM chips that are separate from the processor chip. This is due to two reasons. First, a large capacity main memory requires such a large number of DRAM cells that they cannot all fit on the same chip as the processor. Second, the process technology needed to manufacture DRAM cells (with their capacitors) is not compatible at low cost with the process technology needed to manufacture processor chips. Therefore, placing DRAM and logic together would significantly increase cost in today’s systems. Instead, main memory is implemented as one or more DRAM chips that are dedicated for the large number of DRAM cells. From the perspective of the processor, since DRAM-based main memory is not on the same chip as the processor, it is sometimes referred to as “off-chip” main memory.

A set of electrical wires, called the *memory bus*, connects the processor to the DRAM chips. Within a processor, there is a *memory controller* that communicates with the DRAM chips using the memory bus. In order to access a piece of data from the DRAM chips, the memory controller sends/receives the appropriate electrical signals to/from the DRAM chips through the memory bus. There are three types of signals: *(i)* address, *(ii)* command, and *(iii)* data. The address conveys the location of the data being accessed, the command conveys how the data is being accessed (e.g., read or write), and the data is the actual value of the data itself. Correspondingly, a memory bus is divided into three smaller sets of wires, each of which is dedicated to a specific type of signal: *(i)* address bus, *(ii)* command bus, and *(i)* data bus. Among these three, only the data bus is bi-directional since the memory controller can either send/receive data from the DRAM chips, whereas the address and command buses are uni-directional since only the memory controller sends the address and command to the DRAM chips.

1.5.1 DRAM Organization

As shown in Figure 1.7, a DRAM-based main memory system is logically organized as a hierarchy of *(i)* channels, *(ii)* ranks, and *(iii)* banks. *Banks* are the smallest memory structures that can be accessed in parallel with respect to each other. This is referred to as *bank-level parallelism* [34]. Next, a *rank* is a collection of DRAM chips (and their banks) that operates in lockstep. A DRAM rank typically consists of eight DRAM chips, each of which has eight banks. Since the chips operate in lockstep, the rank has only eight independent banks, each of which is the set of the i^{th} bank across all chips. Banks in different ranks are fully decoupled with respect to their chip-level electrical operation and, consequently, offer better bank-level parallelism than banks in the same rank. Lastly, a *channel* is the collection of all banks that share the same memory bus (address, command, data buses). While banks from the same channel experience contention at the memory bus, banks from different channels can be accessed completely independently of each other. Although the DRAM system offers varying degrees of parallelism at different levels in its organization, two memory requests that access the same bank must be served one after another. To understand why, let us examine the logical organization of a DRAM bank as seen by the memory controller.

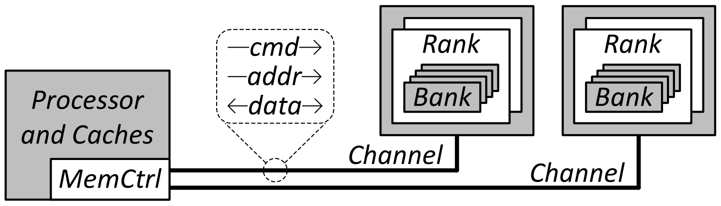
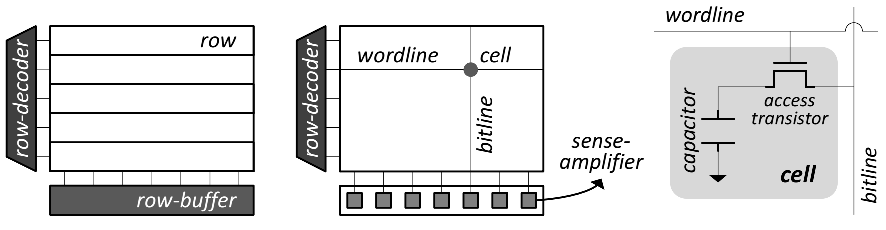


FIGURE 1.7: DRAM organization

Figure 1.8 presents the logical organization of a DRAM bank. A DRAM bank is a two-dimensional array of capacitor-based DRAM cells. It is viewed as a collection of *rows*, each of which consists of multiple *columns*. Therefore, every cell is identified by a pair of addresses: a row address and a column address. Each bank contains a *row-buffer* which is an array of sense-amplifiers. The purpose of a *sense-amplifier* is to read from a cell by reliably detecting the very small amount of electrical charge stored in the cell. When writing to a cell, on the other hand, the sense-amplifier acts as an electrical driver and programs the cell by filling or depleting its stored charge. Spanning a bank in the column-wise direction are wires called the *bitlines*, each of which can



(a) Bank: high-level view (b) Bank: low-level view (c) Cell

FIGURE 1.8: Bank organization

connect a sense-amplifier to any of the cells in the same column. A wire called the *wordline* (one for each row) determines whether or not the corresponding row of cells is connected to the bitlines.

1.5.2 Bank Operation

To serve a memory request that accesses data at a particular row and column address, the memory controller issues three commands to a bank in the order listed below. Each command triggers a specific sequence of events within the bank to access the cell(s) associated with the address.

1. ACTIVATE (issued with a row address): Load the entire row into the row-buffer.

2. READ/WRITE (issued with a column address): From the row-buffer, access the data stored in a column. For a READ, transfer the data out of the row-buffer to the processor. For a WRITE, modify the data in the row-buffer according to the data received from the processor.

3. PRECHARGE: Clear the row-buffer.

Each DRAM command incurs a latency while it is processed by the DRAM chip. Undefined behavior may arise if a command is issued before the previous command is fully processed. To prevent such occurrences, the memory controller must obey a set of *timing constraints* while issuing commands to a DRAM chip [16]. These constraints define when a command becomes ready to be scheduled depending on all other commands issued before it to the same channel, rank, or bank. The exact values of the timing constraints are specified by the DRAM chip's datasheet and are different on a chip-by-chip basis. However, a rule of thumb is that the three DRAM commands (ACTIVATE, READ/WRITE, and PRECHARGE) each take about 15ns [16]. For more information on the organization and operation of a DRAM bank, we refer the reader to Kim et al. [20]

1.5.3 Memory Request Scheduling

When there are many memory requests waiting to access DRAM, the memory controller must choose one of the memory requests to schedule next. The memory controller does so by employing a *memory scheduling algorithm* [40] whose goal, in many current high-performance systems, is to select the most favorable memory request for reducing the overall latency of the memory requests. For example, let us assume there is a memory request waiting to access a row that happens to be loaded in the row-buffer. In this case, since the row is already in the row-buffer, the memory controller can skip the ACTIVATE and PRECHARGE commands and directly proceed to issue the READ/WRITE command. As a result, the memory controller can quickly serve that particular memory request. Such a memory request – that accesses the row in the row-buffer – is called a *row-buffer hit*. Many memory scheduling algorithms exploit the low latency nature of row-buffer hits and prioritize such requests over others [18, 19, 33, 34]. For more information on memory request scheduling, we refer the reader to recent works that explored the topic [18, 19, 33, 34].

1.5.4 Refresh

A DRAM cell stores data as charge on a capacitor. Over time, this charge steadily leaks, causing the data to be lost. That is why DRAM is named “dynamic” RAM, since its charge changes over time. In order to preserve data integrity, the charge in each DRAM cell must be periodically restored or *refreshed*. DRAM cells are refreshed at the granularity of a row by reading it out and writing it back in – which is equivalent to issuing an ACTIVATE and a PRECHARGE to the row in succession.

In modern DRAM chips, all DRAM rows must be refreshed once every 64ms [16], which is called the *refresh interval*. The memory controller internally keeps track of time to ensure that it refreshes all DRAM rows before their refresh interval expires. When the memory controller decides to refresh the DRAM chips, it issues a REFRESH command. Upon receiving a REFRESH command, a DRAM chip internally refreshes a few of its rows by activating and precharging them. A DRAM chip refreshes only a few rows at a time since it has a very large number of rows and refreshing all of them would incur a very large latency. Since a DRAM chip cannot serve any memory requests while it is being refreshed, it is important that the refresh latency is kept short such that no memory request is delayed for too long. So instead of refreshing *all* rows at the end of each 64ms interval, throughout a given 64ms time interval, the memory controller issues many REFRESH commands, each of which triggers the DRAM chip to refresh only a subset of rows. However, the memory controller ensures that REFRESH commands are issued frequently enough such that all rows eventually do become refreshed before 64ms has passed. For more information on DRAM refresh (and methods to reduce its effect on performance and energy), we refer the reader to Liu et al. [27]

1.6 Current and Future Research Issues

The memory system continues to be a major bottleneck in almost all computing systems (especially in terms of performance and energy/power consumption). It is becoming even more of a computing system bottleneck today and looking into the future: more and increasingly diverse processing cores and agents are sharing parts of the memory system; applications that run on the cores are becoming increasingly data and memory intensive; memory is consuming significant energy and power in modern systems; and, there is increasing difficulty scaling the well-established memory technologies, such as DRAM, to smaller technology nodes. As such, managing memory in significantly better ways at all levels of the transformation hierarchy, including both the software and hardware levels, is becoming even more important. Techniques or combinations of techniques that integrate the best ideas cooperatively at multiple levels together appear promising to solve the difficult performance, energy efficiency, correctness, security and reliability problems we face in designing and managing memory systems today. In this section, we briefly discuss some of the major research problems (as we see them) related to caches and main memory, and describe recent potential solution directions. Note that a comprehensive treatment of all research issues is out of the scope of this chapter, and neither is it our intent. For an illustrative treatment of some of the major research issues in memory systems, we refer the reader to Mutlu [32] (`http://users.ece.cmu.edu/~omutlu/pub/onur-ismm-mspc-keynote-june-5-2011-short.pptx`).

1.6.1 Caches

Efficient Utilization. To better utilize the limited capacity of a cache, many replacement policies have been proposed to improve upon the simple LRU policy. The LRU policy is not always the most beneficial across all memory access patterns that have different amounts of locality. For example, in the LRU policy, the most-recently-accessed data is always allocated in the cache even if the data has low-locality (i.e., unlikely to be ever accessed again). To make matters even worse, the low-locality data is unnecessarily retained in the cache for a long time. This is because the LRU policy always *inserts* data into the cache as the most-recently-accessed and evicts the data only after it becomes the least-recently-accessed. As a solution, researchers have been working to develop sophisticated replacement policies using a combination of three approaches. First, when a cache block is allocated in the cache, it should not always be inserted as the most-recently-accessed. Instead, cache blocks with low-locality should be inserted as the least-recently-accessed, so that they are quickly evicted from the cache to make room for other cache blocks that may have more locality (e.g., [15, 35, 41]). Second, when a cache block is evicted from the cache, it should not always be the least-recently-accessed cache block. Ideally, "dead" cache blocks (which will never be accessed in the future) should be evicted – regardless of whether they are least-recently-accessed or not. Third, when choosing the cache block to evict, there should also be consideration for how costly it is to refetch the cache block from main memory. Between two cache blocks, all else being equal, the cache block that is likely to incur the shorter latency to refetch from main memory should be evicted [36].

Quality-of-Service. In a multi-core system, the processor consists of many cores, each of which can independently execute a separate program. In such a system, parts of the memory hierarchy may be shared by some or all of the cores. For example, the last-level cache in a processor is typically shared by all the cores. This is because the last-level cache has a large capacity and, hence, it is expensive to have multiple last-level caches separately for each core. In this case, however, it is important to ensure that the shared last-level cache is utilized by all the cores in a fair manner. Otherwise, a program running on one of the cores may fill up the last-level cache with only its data and evict the data needed by programs running on the other cores. To prevent such occurrences, researchers have proposed mechanisms to provide quality-of-service when a cache is shared by multiple cores. One mechanism is to partition the shared cache among the cores in such a way that each core has a dedicated partition where only the core's data is stored (e.g., [14, 17, 25, 37]). Ideally, the size of a core's partition should be just large enough to hold the data that the core is actively accessing (the core's working set) and not any larger. Furthermore, as the size of a core's working set changes over time, the size of the core's partition should also dynamically expand or shrink in an appropriate manner.

Low Power. Increasing the energy-efficiency of computing systems is one of the key challenges faced by computer architects. Caches are one of the prime targets for energy optimization since they require large numbers of transistors, all of which dissipate power. For example, researchers have proposed to reduce a cache's power consumption by lowering the operating voltage of the cache's transistors [43]. However, this introduces new trade-offs in designing a cache that must be balanced: while a low voltage cache consumes less power, it may also be slower and more prone to errors. Researchers have been examining solutions to enable low-voltage caches that provide acceptable reliability at acceptable cost [43].

1.6.2 Main Memory

Challenges in DRAM Scaling. Primarily due to its low cost-per-bit, DRAM has long been the most popular physical substrate for implementing main memory. In addition, DRAM's cost-per-bit has continuously decreased as DRAM process technology scaled to integrate more DRAM cells into the same area on a semiconductor chip. However, improving DRAM cell density by reducing the cell size, as has been done traditionally, is becoming more difficult due to increased manufacturing complexity/cost and reduced cell reliability. As a result, researchers are examining alternative ways of enhancing the performance and energy-efficiency of DRAM while still maintaining low cost. For example, there have been recent proposals to reduce DRAM access latency [24], to increase DRAM parallelism [20], and to reduce the number of DRAM refreshes [27].

3D-Stacked Memory. A DRAM chip is connected to the processor chip by a memory bus. In today's systems, the memory bus is implemented using electrical wires on a motherboard. But, this has three disadvantages: high power, low bandwidth, and high cost. First, since a motherboard wire is long and thick, it takes a lot of power to transfer an electrical signal through it. Second, since it takes even more power to transfer many electrical signals in quick succession, there is a limit on the bandwidth that a motherboard wire can provide. Third, the processor chip and the DRAM chip each have a stub (called a *pin*) to which either end of a motherboard wire is connected. Unfortunately, pins are expensive. Therefore, a memory bus that consists of many motherboard wires (which requires just as many pins on the chips) increases the cost of the memory system. As a solution, researchers have proposed to stack one more DRAM chips directly on top a processor chip [4, 28] instead of placing them side by side on the motherboard. In such a 3D-stacked configuration, the processor chip can communicate directly with the DRAM chip(s), thereby eliminating the need for motherboard wires and pins.

Emerging Technology. DRAM is by far the most dominant technology for implementing main memory. However, there has been research to develop alternative main memory technologies that may replace DRAM or be used in conjunction with DRAM. For example, *PCM* (phase-change-memory, pronounced "pee-cee-em") is a technology in which data is stored as a resistance value. A PCM cell consists of a small crystal whose resistance value can be changed by applying heat to it at different rates. If heat is abruptly applied by a short burst of high current, then the crystal is transformed into the amorphous state, which has high resistance. On the other hand, if heat is steadily applied by a long burst of low current, then the crystal is transformed into the crystalline state, which has low resistance. PCM has important advantages that make it an attractive candidate for main memory [21, 38]: non-volatility (i.e., data is not lost when the power is turned off and there is no need to refresh PCM cells) and scalability (i.e., compared to DRAM, it may be easier to make smaller PCM cells in the future). However, PCM also has the disadvantages of large latency (especially for writes) and limited write-endurance (i.e., the PCM cell becomes unreliable beyond a certain number of writes), the solution to which is the topic of on-going research. To overcome the shortcomings of different technologies, several recent works have examined the use of multiple different technologies (e.g., PCM and DRAM together) as part of main memory, an approach called *hybrid memory* or *heterogeneous memory* [7, 29, 38, 45]. The key challenge in hybrid memory systems is to devise effective algorithms that place data in the appropriate technology such that the system can exploit the advantages of each technology while hiding the disadvantages of each [29, 45].

Quality-of-Service. Similar to the last-level cache, main memory is also shared by all the cores in a processor. When the cores contend to access main memory, their accesses may interfere with each other and cause significant delays. In the worst case, a memory-intensive core may continuously access main memory in such a way that all the other cores are denied service from main memory [30]. This would detrimentally degrade the performance of not only those particular cores, but also of the entire computing system. To address this problem, researchers have proposed mechanisms that provide quality-of-service to each core when accessing shared main memory. For example, memory request scheduling algorithms can ensure that memory requests from all the cores are served in a fair manner [18, 19, 33, 34]. Another approach is for the user to explicitly specify memory service requirements of a program to the memory controller so that the memory scheduling algorithm can subsequently guarantee that those requirements are met [42]. Other approaches to quality-of-service include mechanisms proposed to map the data of those applications that significantly interfere with each other to different memory channels [31] and mechanisms proposed to *throttle down* the request rate of the processors that cause significant interference to other processors [8]. Request scheduling mechanisms that prioritize bottleneck threads in parallel applications have also been proposed [10]. The QoS problem gets exacerbated when the processors that share the main memory are different, e.g., when main memory is shared by a CPU consisting of multiple processors and a GPU, and recent research has started to examine solutions to this [1]. Finally, providing QoS and high performance in the presence of different types of memory requests from multiple processing cores, such as speculative *prefetch* requests that aim to fetch the data from memory before it is needed, is a challenging problem that recent research has started providing solutions for [22, 23, 11, 9].

1.7 Summary

The memory system is a critical component of a computing system. It serves as the repository of data from where the processor (or processors) can access data. An ideal memory system would have both high performance and large capacity. However, there exists a fundamental trade-off relationship between the two: it is possible to achieve either high performance or large capacity, but not both at the same time in a cost-effective manner. As a result of the trade-off, a memory system typically consists of two components: caches (which are small but relatively fast-to-access) and main memory (which is large but relatively slow-to-access). Multiple caches and a single main memory, all of which strike a different balance between performance and capacity, are combined to form a memory hierarchy. The goal of the memory hierarchy is to provide the high performance of a cache at the large capacity of main memory. The memory system is co-operatively managed by both the operating system and the hardware.

This chapter provided an introductory level description of memory systems employed in modern computing systems, focusing especially on how the memory hierarchy, consisting of caches and main memory, is organized and managed. The memory system continues to be an even more critical bottleneck going into the future, as described in Section 1.6. Many problems abound, yet the authors of this chapter remain confident that promising solutions, some of which are also described in Section 1.6, will also abound and hopefully prevail.

Bibliography

[1] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *International Symposium on Computer Architecture*, 2012.

[2] J.-L. Baer and W.-H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *International Symposium on Computer architecture*, 1988.

[3] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Syst. J.*, 5(2), June 1966.

[4] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H Loh, Don McCaule, Pat Morrow, Donald W Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. Die Stacking (3D) Microarchitecture. In *International Symposium on Microarchitecture*, 2006.

[5] F. J. Corbató. A Paging Experiment with the Multics System. *In Honor of P. M. Morse*, 1969.

[6] Robert H. Dennard. Field-Effect Transistor Memory. U.S. Patent Number 3387286, 1968.

[7] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. PDRAM: A Hybrid PRAM and DRAM Main Memory System. In *Design Automation Conference*, 2009.

[8] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *Architectural Support for Programming Languages and Operating Systems*, 2010.

[9] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Prefetch-Aware Shared Resource Management for Multi-Core Systems. In *International Symposium on Computer Architecture*, 2011.

[10] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A. Joao, Onur Mutlu, and Yale N. Patt. Parallel Application Memory Scheduling. In *International Symposium on Microarchitecture*, 2011.

[11] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N Patt. Coordinated Control of Multiple Prefetchers in Multi-Core Systems. In *International Symposium on Microarchitecture*, 2009.

[12] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.*, 38(12), Dec 1989.

[13] Intel. Intel 64 and IA-32 Architectures Software Developers Manual, Aug 2012.

[14] Ravi Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *International Conference on Supercomputing*, 2004.

[15] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr., and Joel Emer. Adaptive Insertion Policies for Managing Shared Caches. In *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[16] Joint Electron Devices Engineering Council (JEDEC). DDR3 SDRAM Standard (JESD79-3F), 2012.

[17] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[18] Yoongu Kim, Dongsu Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *International Symposium on High Performance Computer Architecture*, 2010.

[19] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *International Symposium on Microarchitecture*, 2010.

[20] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *International Symposium on Computer Architecture*, 2012.

[21] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *International Symposium on Computer Architecture*, 2009.

[22] Chang J Lee, Onur Mutlu, Veynu Narasiman, and Yale N Patt. Prefetch-Aware DRAM Controllers. In *International Symposium on Microarchitecture*, 2008.

[23] Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N Patt. Improving Memory Bank-Level Parallelism in the Presence of Prefetching. In *International Symposium on Microarchitecture*, 2009.

[24] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *International Symposium on High Performance Computer Architecture*, 2013.

[25] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *International Symposium on High Performance Computer Architecture*, 2008.

[26] J. S. Liptay. Structural Aspects of the System/360 Model 85: II The Cache. *IBM Syst. J.*, 7(1), Mar 1968.

[27] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *International Symposium on Computer Architecture*, 2012.

[28] Gabriel H Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *International Symposium on Computer Architecture*, 2008.

[29] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *IEEE Computer Architecture Letters*, 11(2), July 2012.

[30] Thomas Moscibroda and Onur Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security Symposium*, 2007.

[31] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *International Symposium on Microarchitecture*, 2011.

[32] Onur Mutlu. Memory Systems in the Many-Core Era: Challenges, Opportunities, and Solution Directions. In *International Symposium on Memory Management*, 2011. `http://users.ece.cmu.edu/~omutlu/pub/onur-ismm-mspc-keynote-june-5-2011-short.pptx`.

[33] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *International Symposium on Microarchitecture*, 2007.

[34] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *International Symposium on Computer Architecture*, 2008.

[35] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *International Symposium on Computer Architecture*, 2007.

[36] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A Case for MLP-Aware Cache Replacement. In *International Symposium on Computer Architecture*, 2006.

[37] Moinuddin K Qureshi and Yale N Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *International Symposium on Microarchitecture*, 2006.

[38] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *International Symposium on Computer Architecture*, 2009.

[39] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *International Symposium on Computer Architecture*, 2005.

[40] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory Access Scheduling. In *International Symposium on Computer Architecture*, 2000.

[41] Vivek Seshadri, Onur Mutlu, Michael A Kozuch, and Todd C Mowry. The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing. In *International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[42] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *International Symposium on High Performance Computer Architecture*, 2013.

[43] Chris Wilkerson, Hongliang Gao, Alaa R Alameldeen, Zeshan Chishti, Muhammad Khellah, and Shih-Lien Lu. Trading off Cache Capacity for Reliability to Enable Low Voltage Operation. In *International Symposium on Computer Architecture*, 2008.

[44] M. V. Wilkes. Slave Memories and Dynamic Storage Allocation. *Electronic Computers, IEEE Transactions on*, EC-14(2), 1965.

[45] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A. Harding, and Onur Mutlu. Row buffer locality aware caching policies for hybrid memories. In *International Conference on Computer Design*, 2012.