10
The Memory System

Up till now, we have considered the memory system to be one large array of bytes. This abstraction was good enough for designing an instruction set, studying assembly language, and even for designing a basic processor with a complicated pipeline. However, from a practical standpoint, this abstraction will need to be further refined to design a fast memory system. In our basic *SimpleRisc* pipeline presented in Chapter 8 and 9, we have assumed that it takes 1 cycle to access both data and instruction memory. We shall see in this chapter, that this is not always true. In fact, we need to make significant optimisations in the memory system to come close to the ideal latency of 1 cycle. We need to introduce the notion of a "cache" and a hierarchical memory system to solve the dual problems of having large memory capacity, and low latency.

Secondly, up till now we have been assuming that only one program runs on our system. However, most processors typically run multiple programs on a time shared basis. For example, if there are two programs, A and B, a modern desktop or laptop typically runs program A for a couple of milliseconds, executes B for a few milliseconds, and subsequently switches back and forth. In fact as your author is writing this book, there are a host of other programs running on his system such as a web browser, an audio player, and a calendar application. In general, a user does not perceive any interruptions, because the time scale at which the interruptions happen is much lower than what the human brain can perceive. For example, a typical video displays a new picture 30 times every second, or alternatively one new picture every 33 milliseconds. The human brain creates the illusion of a smoothly moving object by piecing the pictures together. If the processor finishes the job of processing the next picture in a video sequence, before 33 milliseconds, then it can execute a part of another program. The human brain will not be able to tell the difference. The point here is that without our knowledge, the processor in co-operation with the operating system switches between multiple programs many many times a second. The *operating system* is itself a specialised program that helps the processor manage itself, and other programs. Windows and Linux are examples of popular operating systems.

We shall see that we require special support in the memory system to support multiple

programs. If we do not have this support, then multiple programs can overwrite each other's data, which is not desired behavior. Secondly, we have been living with the assumption that we have practically an infinite amount of memory. This is also not true. The amount of memory that we have is finite, and it can get exhausted by large memory intensive programs. Hence, we should have a mechanism to still run such large programs. We shall introduce the concept of virtual memory to solve both of these issues – running multiple programs, and handling large memory intensive programs.

To summarise, we observe that we need to design a memory system that is fast, and is flexible enough to support multiple programs with very large memory requirements.

10.1 Overview

10.1.1 Need for a Fast Memory System

Let us now look at the technological requirements for building a fast memory system. We have seen in Chapter 6 that we can design memory elements with four kinds of basic circuits – latches, SRAM cells, CAM cells and DRAM cells. There is a tradeoff here. Latches and SRAM cells are much faster than DRAM or CAM cells. However, as compared to a DRAM cell, a latch, CAM or SRAM cell is an order of magnitude larger in terms of area, and also consumes much more power. We observe that a latch is designed to read in and read out data at a negative clock edge. It is a fast circuit that can store and retrieve data in a fraction of a clock cycle. On the other hand, an SRAM cell is typically designed to be used as a part of a large array of SRAM cells along with a decoder and sense amplifiers. With this additional overhead, an SRAM cell is typically slower than a typical edge triggered latch. In comparison, CAM cells are best for memories that are content associative, and DRAM cells are best for memories that have very large capacities.

Now, our *SimpleRisc* pipeline assumes that memory accesses take 1 cycle. To satisfy this requirement, we need to build our entire memory from latches, or small arrays of SRAM cells. Table 10.1 shows the size of a typical latch, SRAM cell, and DRAM cell as of 2012.

Cell type	Area	Typical latency (array of cells)
Master Slave D flip flop	0.8 μm^2	fraction of a cycle
SRAM cell	0.08 μm^2	1-5 cycles
DRAM Cell	0.005 μm^2	50-200 cycles

Table 10.1: Sizes of a Latch, SRAM cell, and DRAM cell

We observe that a typical latch (master slave D flip flop) is 10 times larger than an SRAM cell, which in turn is around 16 times larger than a DRAM cell. This means that given a certain amount of silicon, we can save 160 times more data if we use DRAM cells. However, DRAM memory is also 200 times slower (if we consider a representative array of DRAM cells). Clearly, there is a tradeoff between capacity, and speed. The sad part is that we actually need both.

Let us consider the issue of capacity first. Due to several constraints in technology and manufacturability, as of 2012, it is not possible to manufacture chips with an area more than

400-500mm^2 [ITRS, 2011]. Consequently, the total amount of memory that we can have on chip is limited. It is definitely possible to supplement the amount of available memory with additional chips exclusively containing memory cells. Keep in mind that off-chip memory is slow, and it takes tens of cycles for the processor to access such memory modules. To achieve our goal of having a 1-cycle memory access, we need to use the relatively faster on-chip memory most of the time. Here, also our options are limited. We cannot afford to have a memory system consisting exclusively of latches. For a large number of programs, we will not be able to fit all our data in memory. For example, modern programs typically require hundreds of megabytes of memory. Moreover, some large scientific programs require gigabytes of memory. Second, it is difficult to integrate large DRAM arrays along with a processor on the same chip due to technological constraints. Hence, designers are compelled to use large SRAM arrays for on-chip memories. As shown in Table 10.1 SRAM cells(arrays) are much larger than DRAM cells(arrays), and thus have much less capacity.

There is a conflicting requirement of latency. Let us assume that we decide to maximise storage, and make our memory entirely consisting of DRAM cells. Let us assume a 100 cycle latency for accessing DRAM. If we assume that a third of our instructions are memory instructions, then the effective CPI of a perfect 5 stage *SimpleRisc* pipeline is calculated to be $1 + 1/3 \times (100 - 1) = 34$. The point to note is that our CPI increases by 34X, which is completely unacceptable.

Hence, we need to make an equitable tradeoff between latency and storage. We want to store as much of data as possible, but not at the cost of a very low IPC. Unfortunately, there is no way out of this situation, if we assume that our memory accesses are completely random. If there is some pattern in memory accesses, then we can possibly do something better such that we can get the best of both worlds – high storage capacity, and low latency.

10.1.2 Memory Access Patterns

Before considering the technical topic of patterns in memory accesses, let us consider a simple practical problem that your author is facing at this point of time. He unfortunately, has a lot of books on his desk that are not organised. Not only are these books cluttering up his desk, it is also hard to search for a book when required. Hence, he needs to organise his books better and also keep his desk clean. He observes that he does not require all the books all the time. For example, he needs books on computer architecture very frequently; however, he rarely reads his books on distributed systems. Hence, it makes sense for him to move his books on distributed systems to the shelf beside his desk. Unfortunately, it is a small shelf, and there are still a lot of books on his desk. He observes that he can further classify the books in the small shelf. He has some books on philosophy that he never reads. These can be moved to the large cabinet in the corner of the room. This will create more space in the shelf, and also help him clean up his desk. What is the fundamental insight here? It is that your author does not read all his books with the same frequency. There are some books that he reads very frequently; hence, they need to be on his desk. Then there is another class of books that he reads infrequently; hence, they need to be in the small shelf beside his desk. Lastly, he has a large number of books that he reads extremely infrequently. He can safely keep them in the large cabinet. **Pattern 1:He reads a small set of books very frequently, and the rest of the books rather infrequently.** Hence, if he keeps the frequently accessed set of books on computer architecture

on his desk, and the large infrequent set of books in the shelf and the cabinet, he has solved his problems.

Well, not quite. This was true for last semester, when he was teaching the computer architecture course. However, in the current semester, he is teaching a course on distributed systems. Hence, he does not refer to his architecture books anymore. It thus makes sense for him to bring his distributed systems books to his desk. However, there is a problem. What happens to his architecture books that are already there on his desk. Well, the simple answer is that they need to be moved to the shelf and they will occupy the slots vacated by the distributed systems books. In the interest of time, it makes sense for your author to bring a set of distributed systems books on to his desk, because in very high likelihood, he will need to refer to numerous books in that area. It does not make sense to fetch just one book on distributed systems. Hence, as a general rule we can conclude that if we require a certain book, then most likely we will require other books in the same subject. **Pattern 2:If your author requires a certain book, then most likely he will require other books in the same subject area in the near future.**

We can think of patterns 1 and 2, as general laws that are applicable to everybody. Instead of books, if we consider TV channels, then also both the patterns apply. We do not watch all TV channels equally frequently. Secondly, if a user has tuned in to a news channel, then most likely she will browse through other news channels in the near future. In fact this is how retail stores work. They typically keep spices and seasonings close to vegetables. This is because it is highly likely that a user who has just bought vegetables will want to buy spices also. However, they keep bathroom supplies and electronics far away.

Pattern 1 is called *temporal locality*. This means that users will tend to reuse the same item in a given time interval. Pattern 2 is called *spatial locality*. It means that if a user has used a certain item, then she will tend to use similar items in the near future.

Definition 97

Temporal Locality *It is a concept that states that if a resource is accessed at some point of time, then most likely it will be accessed again in a short time interval.*

Spatial Locality *It is a concept that states that if a resource is accessed at some point of time, then most likely similar resources will be accessed in the near future.*

The question that we need to ask is – "Is there temporal and spatial locality in memory accesses?". If there is some degree of temporal and spatial locality, then we can possibly do some critical optimisations that will help us solve the twin problems of large memory requirement, and low latency. In computer architecture, we typically rely on such properties such as temporal and spatial locality to solve our problems.

10.1.3 Temporal and Spatial Locality of Instruction Accesses

The standard approach for tackling this problem, is to measure and characterise locality in a representative set of programs such as the SPEC benchmarks(see Section 9.9.4). Let us first start out by dividing memory accesses into two broad types – instruction and data. Instruction accesses are much easier to analyse informally. Hence, let us look at it first.

Let us consider a typical program. It has assignment statements, decision statements (if,else), and loops. Most of the code in large programs is part of loops or some pieces of common code. There is a standard rule of thumb in computer architecture, which states that 90% of the code runs for 10% of time, and 10% of the code runs for 90% of the time. Let us consider a word processor. The code to process the user's input, and show the result on the screen runs much more frequently than the code for showing the help screen. Similarly, for scientific applications, most of the time is spent in a few loops in the program. In fact for most common applications, we find this pattern. Hence, computer architects have concluded that temporal locality for instruction accesses holds for an overwhelming majority of programs.

Let us now consider spatial locality for instruction accesses. If there are no branch statements, then the next program counter is the current program counter plus 4 bytes for an ISA such as *SimpleRisc* . We consider two accesses to be "similar", if their memory addresses are close to each other. Clearly, we have spatial locality here. A majority of the instructions in programs are non-branches; hence, spatial locality holds. Moreover, a nice pattern in branches in most programs is that the branch target is actually not very far away. If we consider a simple *if-then* statement or *for* loop then the distance of the branch target is equal to the length of the loop or the *if* part of the statement. In most programs this is typically 10 to 100 instructions long, definitely not thousands of instructions long. Hence, architects have concluded that instruction memory accesses exhibit a good amount of spatial locality also.

The situation for data accesses is slightly more complicated; however, not very different. For data accesses also we tend to reuse the same data, and access similar data items. Let us look at this in more detail.

10.1.4 Characterising Temporal Locality

Let us describe a method called the method of stack distances to characterise temporal locality in programs.

Stack Distance

We maintain a stack of accessed data addresses For each memory instruction (load/store), we search for the corresponding address in the stack. The position at which the entry is found (if found) is termed the "stack distance". Here, the distance is measured from the top of the stack. The top of the stack has distance equal to zero, whereas the 100^{th} entry has a stack distance equal to 99. Whenever, we detect an entry in the stack we remove it, and push it to the top of the stack.

If the memory address is not found, then we make a new entry and push it to the top of the stack. Typically, the depth of the stack is bounded. It has length, L. If the number of entries in the stack exceeds L because of the addition of a new entry, then we need to remove the entry at the bottom of the stack. Secondly, while adding a new entry, the stack distance

is not defined. Note that since we consider bounded stacks, there is no way of differentiating between a new entry, and an entry that was there in the stack, but had to be removed because it was at the bottom of the stack. Hence, in this case we take the stack distance to be equal to L (bound on the depth of the stack).

Note that the notion of stack distance gives us an indication of temporal locality. If the accesses have high temporal locality, then the mean stack distance is expected to be lower. Conversely, if memory accesses have low temporal locality, then the mean stack distance will be high. We can thus use the distribution of stack distances as a measure of the amount of temporal locality in a program.

Experiment to Measure Stack Distance

We perform a simple experiment with the SPEC2006 benchmark, Perlbench, which runs different Perl programs [1]. We maintain counters to keep track of the stack distance. The first million memory accesses serve as a warm-up period. During this time the stack is maintained, but the counters are not incremented. For the next million memory accesses, the stack is maintained, and the counters are also incremented. Figure 10.1 shows a histogram of the stack distance. The size of the stack is limited to 1000 entries. It is sufficient to capture an overwhelming majority of memory accesses.

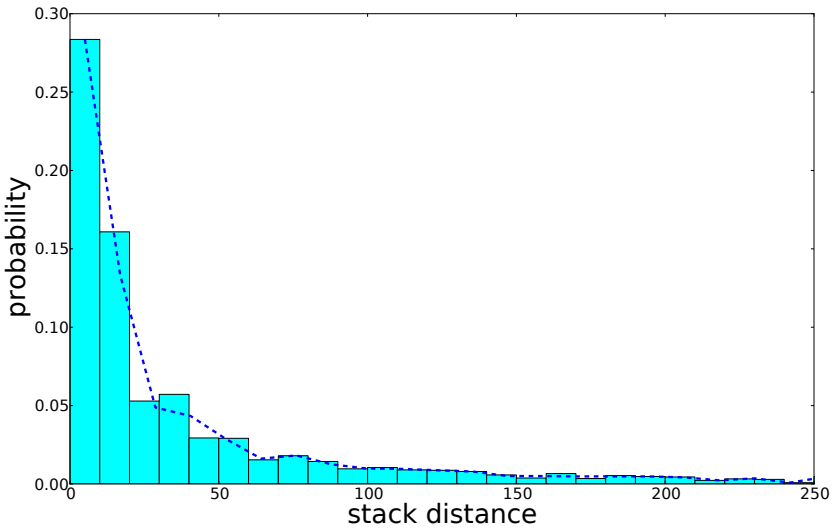


Figure 10.1: Stack distance distribution

We observe that most of the accesses have a very low stack distance. A stack distance between 0-9 is the most common value. Approximately 27% of all the accesses are in this bin. In fact, more than two thirds of the memory accesses have a stack distance less than 100. Beyond 100, the distribution tapers off, yet remains fairly steady. The distribution of stack distances is

[1] Dataset size 'ref', input 'split-mail'

typically said to follow a heavy tailed distribution. This means that the distribution is heavily skewed towards smaller stack distances; however, large stack distances are not uncommon. The tail of the distribution continues to be non-zero for large stack distances. We observe a similar behavior here.

Trivia 3 *Researchers have tried to approximate the stack distance using the log-normal distribution.*

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(ln(x)-\mu)^2}{2\sigma^2}}$$

10.1.5 Characterising Spatial Locality

Address Distance

Akin to stack distance, we define the term *address distance*. The i^{th} address distance is the difference in the memory address of the i^{th} memory access, and the closest address in the set of the last K memory accesses. Here, a memory access can be either a load or a store. There is an intuitive reason for defining address distance in this manner. Programs typically access different regions of main memory in the same time interval. For example, an operation on arrays, accesses an array item, then accesses some constants, performs an operation, saves the result, and then moves on the next array entry using a *for* loop. There is clearly spatial locality here, in the sense that consecutive iterations of a *for* loop access proximate addresses in an array. However, to quantify it, we need to search for the closest access (in terms of memory addresses) over the last K accesses. Here, K is the number of memory accesses in each iteration of the enclosing loop. We can readily observe that in this case that the address distance turns out to be a small value, and is indicative of high spatial locality. However, K needs to be well chosen. It should not be too small, nor too large. We have empirically found $K = 10$ to be an appropriate value for a large set of programs.

To summarise, we can conclude that if the average address distance is small, then it means that we have high spatial locality in the program. The program tends to access nearby memory addresses with high likelihood in the same time interval. Conversely, if the address distances are high, then the accesses are far apart from each other, and the program does not exhibit spatial locality.

Experiment to Characterise Address Distance

Here, we repeat the same experiment as described in Section 10.1.4 with the SPEC2006 benchmark, Perlbench. We profile the address distance distribution for the first 1 million accesses. Figure 10.2 shows the address distance distribution.

Here also, more than a quarter of the accesses have an address distance between -5 and +5, and more than two thirds of the accesses have an address distance between -25 and +25. Beyond ±50, the address distance distribution tapers off. Empirically, this distribution also has a heavy tailed nature.

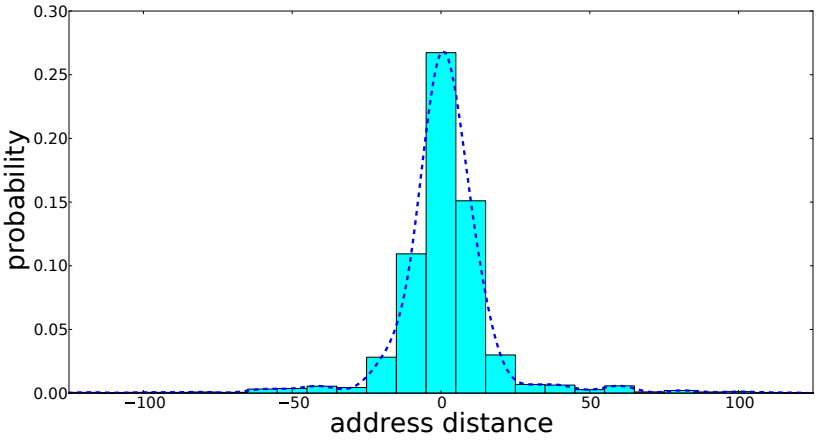


Figure 10.2: Address distance distribution

10.1.6 Utilising Spatial and Temporal Locality

Section 10.1.4 and 10.1.5 showed the stack and address distance distributions for a sample program. Similar experiments have been performed for thousands of programs that users use in their daily lives. These programs include computer games, word processors, databases, spreadsheet applications, weather simulation programs, financial applications, and software applications that run on mobile computers. Almost all of them exhibit a very high degree of temporal and spatial locality. In other words, temporal and spatial locality, are basic human traits. Whatever we do, including fetching books, or writing programs, these properties tend to hold. Note that these are just mere empirical observations. It is always possible to write a program that does not exhibit any form of temporal and spatial locality. Additionally, it is always possible to find regions of code in commercial programs that do not exhibit these properties. However, these examples are exceptions. They are not the norm. We need to design computer systems for the norm, and not for the exceptions. This is how we can boost performance for a large majority of programs that users are expected to run.

From now on, let us take temporal and spatial locality for granted, and see what can be done to boost the performance of the memory system, without compromising on storage capacity. Let us look at temporal locality first.

10.1.7 Exploiting Temporal Locality – Hierarchical Memory System

Let us reconsider the way in which we tried to exploit temporal locality for our simple example with books. If your author decides to look at a new topic, then he brings the set of books associated with that topic to his desk. The books that were already there on his desk, are moved to the shelf, and to create space in the shelf, some books are shifted to the cabinet. This

behavior is completely consistent with the notion of stack distance as shown in Figure 10.1.

We can do the same with memory systems also. Akin to a desk, shelf, and cabinet, let us define a storage location for memory values. Let us call it a *cache*. Each entry in the cache conceptually contains two fields – memory address, and value. Like your author's office, let us define a hierarchy of caches as shown in Figure 10.3.

Definition 98
A cache *contains a set of values for different memory locations.*

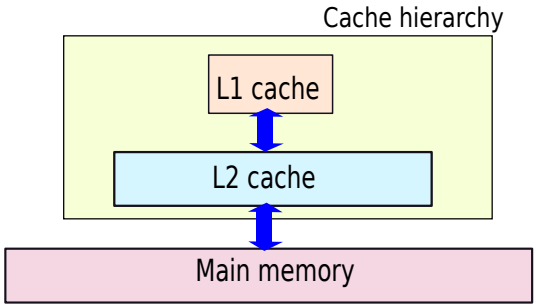


Figure 10.3: Memory hierarchy

Definition 99 *The* main memory *(*physical memory*) is a large DRAM array that contains values for all the memory locations used by the processor.*

The L1 cache corresponds to the desk, the L2 cache corresponds to the shelf, and the main memory corresponds to the cabinet. The L1 cache is typically a small SRAM array (8-64 KB). The L2 cache is a larger SRAM array (128 KB - 4 MB). Some processors such as the Intel Sandybridge processor have another level of caches called the L3 cache (4MB+). Below the L2/L3 cache, there is a large DRAM array containing all the memory locations. This is known as the *main memory* or *physical memory*. Note that in the example with books, a book could either exclusively belong to the shelf or the cabinet. However, in the case of memory values, we need not follow this rule. In fact, we shall see later that it is easier to maintain a subset of values of the L2 cache in the L1 cache, and so on. This is known as a system with inclusive caches. We thus have – $values(L1) \subset values(L2) \subset values(main\ memory)$ – for an inclusive cache hierarchy. Alternatively, we can have exclusive caches, where a higher level cache does not necessarily contain a subset of values in the lower level cache. Inclusive caches are by far used universally in all processors. This is because of the ease of design, simplicity, and some subtle correctness issues that we shall discuss in Chapter 11. There are some research level

proposals that advocate exclusive caches. However, their utility for general purpose processors has not been established as of 2012.

Definition 100 *A memory system in which the set of memory values contained in the cache at the n^{th} level is a subset of all the values contained in the cache at the $(n+1)^{th}$ level, is known as an* inclusive *cache hierarchy. A memory system that does not follow strict inclusion is referred to as an* exclusive *cache hierarchy.*

Let us now consider the cache hierarchy as shown in Figure 10.3. Since the L1 cache is small, it is faster to access. The access time is typically 1-2 cycles. The L2 cache is larger and typically takes 5-15 cycles to access. The main memory is much slower because of its large size and use of DRAM cells. The access times are typically very high and are between 100-300 cycles. The memory access protocol is similar to the way your author accesses his books.

The memory access protocol is as follows. Whenever, there is a memory access (load or store), the processor first checks in the L1 cache. Note that each entry in the cache conceptually contains both the memory address and value. If the data item is present in the L1 cache, we declare a **cache hit**, otherwise we declare a **cache miss**. If there is a cache hit, and the memory request is a read, then we need to just return the value to the processor. If the memory request is a write, then the processor writes the new value to the cache entry. It can then propagate the changes to the lower levels, or resume processing. We shall look at these the different methods of performing a cache write in detail, when we discuss different write policies in Section 10.2.3. However, if there is a cache miss, then further processing is required.

Definition 101

Cache hit *Whenever a memory location is present in a cache, the event is known as a* cache hit*.*

Cache miss *Whenever a memory location is not present in a cache, the event is known as a* cache miss*.*

In the event of an L1 cache miss, the processor needs to access the L2 cache and search for the data item. If an item is found (cache hit), then the protocol is the same as the L1 cache. Since, we consider inclusive caches in this book, it is necessary to fetch the data item to the L1 cache. If there is an L2 miss, then we need to access the lower level. The lower level can be another L3 cache, or can be the main memory. At the lowest level, i.e., the main memory, we are guaranteed to not have a miss, because we assume that the main memory contains an entry for all the memory locations.

Performance Benefit of a Hierarchical Memory System

Instead of having a single flat memory system, processors use a hierarchical memory system to maximise performance. A hierarchical memory system is meant to provide the illusion of a large memory with an ideal single cycle latency.

Example 119 *Find the average memory access latency for the following configurations.*

Configuration 1		
Level	*Miss Rate(%)*	*Latency*
L1	*10*	*1*
L2	*10*	*10*
Main Memory	*0*	*100*
Configuration 2		
Main Memory	*0*	*100*

Answer: *Let us consider the first configuration. Here, 90% of the accesses hit in the L1 cache. Hence, their memory access time is 1 cycle. Note that even the accesses that miss in the L1 cache still incur the 1 cycle delay, because we do not know if an access will hit or miss in the cache. Subsequently, 90% of the accesses that go to the L2 cache hit in the cache. They incur a 10-cycle delay. Finally, the remaining accesses (1%) hit in the main memory, and incur an additional delay. The average memory access time(T) is thus:*

$$T = 1 + 0.1 * (10 + 0.1 * 100) = 1 + 1 + 1 = 3$$

Thus, the average memory latency of a hierarchical memory system such as configuration 1 is 3 cycles.

Configuration 2 is a flat hierarchy, which uses the main memory for all its accesses. The average memory access time is 100 cycles.

There is thus a speedup of 100/3 = 33.3 times using a hierarchical memory system.

Let us consider an example (see Example 119). It shows that the performance gain using a hierarchical memory system is 33.33 times that of a flat memory system with a single level hierarchy. The performance improvement is a function of the hit rates of different caches and their latencies. Moreover, the hit rate of a cache is dependent on the stack distance profile of the program, and the cache management policies. Likewise the cache access latency is dependent on the cache manufacturing technology, design of the cache, and the cache management schemes. We need to mention that optimising cache accesses has been a very important topic in computer architecture research for the past two decades. Researchers have published thousands of papers in this area. We shall only cover some basic mechanisms in this book. The interested reader can take a look at Section 10.5.2 for appropriate references.

10.1.8 Exploiting Spatial Locality – Cache Blocks

Let us now consider spatial locality. We observe in Figure 10.2 that a majority of accesses have an address distance within ±25 bytes. Recall that the address distance is defined as the difference in memory addresses between the current address and the closest address among the last K addresses. The address distance distribution suggests that if we group a set of memory locations into one block, and fetch it at one go from the lower level, then we can increase the number of cache hits because there is a high degree of spatial locality in accesses. This approach is similar to the way we decided to fetch all the architecture books at the same time from the shelf in Section 10.1.2.

Consequently, almost all processors create blocks of contiguous addresses, and the cache treats each block as an atomic unit. The entire block is fetched at once from the lower level, and also an entire block is evicted from the cache if required. A cache block is also known as a *cache line*. A typical cache block or a line is 32-128 bytes long. For ease of addressing, its size needs to be a strict power of 2.

Definition 102
A cache block or a line is a contiguous set of memory locations. It is treated as an atomic unit of data in a cache.

Thus, we need to slightly redefine the notion of a cache entry. Instead of having an entry for each memory address, we have a separate entry for each cache line. Note that in this book, we shall use the terms cache line and block synonymously. Also note that it is not necessary to have the same cache line size in the L1 cache and the L2 cache. They can be different. However, for maintaining the property of inclusiveness of caches, and minimising additional memory accesses, it is typically necessary to use an equal or larger block size in the L2 cache as compared to the L1 cache.

Way Point 9
Here is what we have learnt up till now.

1. *Temporal and spatial locality are properties inherent to most human actions. They apply equally well to reading books and writing computer programs.*
2. *Temporal locality can be quantified by the* stack distance*, and spatial locality can be quantified by the* address distance*.*
3. *We need to design memory systems to take advantage of temporal and spatial locality.*
4. *To take advantage of temporal locality, we use a hierarchical memory system consisting of a set of caches. The L1 cache is typically a small and fast structure that is meant to satisfy most of the memory accesses quickly. The lower level of the caches store larger amounts of data, are accessed infrequently, and have larger access times.*

5. *To take advantage of spatial locality, we group sets of contiguous memory locations into blocks (also known as lines). A block is treated as an atomic unit of data in a cache.*

Given that we have studied the requirements of a cache qualitatively, we shall proceed to discuss the design of caches.

10.2 Caches

10.2.1 Overview of a Basic Cache

Let us consider a cache as a black box as shown in Figure 10.4. In the case of a load operation, the input is the memory address, and the output is the value of the memory location if there is a cache hit. We envision the cache having a status line that indicates if the request suffered a hit or miss. If the operation is a store, then the cache takes two inputs – memory address, and value. The cache stores the value in the entry corresponding to the memory location if there is a cache hit. Otherwise, it indicates that there is a cache miss.

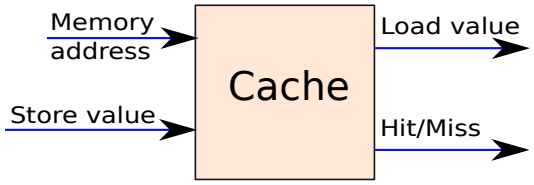


Figure 10.4: A cache as a black box

Let us now look at methods to practically implement this black box. We shall use an SRAM array as the building block (see Section 6.4). The reader might wish to revisit that section to recapitulate her knowledge on memory structures.

To motivate a design, let us consider an example. Let us consider a 32-bit machine with a block size of 64 bytes. In this machine, we thus have 2^{26} blocks. Let the size of the L1 cache be 8 KB. It contains 2^7 or 128 blocks. We can thus visualise the L1 cache at any point of time as a very small subset of the entire memory address space. It contains at the most 128 out of 2^{26} blocks. To find out if a given block is there in the L1 cache, we need to see if any of the 128 entries contains it.

We assume that our L1 cache, is a part of a memory hierarchy. The memory hierarchy as a whole supports two basic requests – *read* and *write*. However, we shall see that at the level of a cache, we require many basic operations to implement these two high level operations.

Basic Cache Operations

Akin to a memory address, let us define a block address as the 26 MSB bits of the memory address. The first problem is to find if a block with the given block address is present in the

cache. We need to perform a *lookup operation* that returns a pointer to the block if it is present in the cache. If the block is present in the cache then we can declare a cache hit and service the request. For a cache hit, we need two basic operations to service the request namely *data read*, and *data write*. They read or write the contents of the block, and require the pointer to the block as an argument.

If there is a cache miss, then we need to fetch the block from the lower levels of the memory hierarchy and insert it in the cache. The procedure of fetching a block from the lower levels of the memory hierarchy, and inserting it into a cache, is known as a *fill* operation. The *fill* operation is a complex operation, and uses many atomic sub-operations. We need to first send a load request to the lower level cache to fetch the block, and then we need to insert in into the L1 cache.

The process of insertion is also a complex process. We need to first check, if we have space to insert a new block in a given set of blocks. If we have sufficient space in a set, then we can populate one of the entries using an *insert* operation. However, if all the locations at which we want to insert a block in the cache are already busy, then we need to evict an already existing block from the cache. We thus need to invoke a *replace* operation to find the cache block that needs to be evicted. Once, we have found an appropriate candidate block for replacement, we need to evict it from the cache using an *evict* operation.

Thus, to summarise the discussion up till now, we can conclude that we broadly need these basic operations to implement a cache – *lookup*, *data read*, *data write*, *insert*, *replace*, and *evict*. The *fill* operation is just a sequence of *lookup*, *insert*, and *replace* operations at different levels of the memory hierarchy. Likewise, the *read* operation is either primarily a *lookup* operation, or the combination of a *lookup* and *fill* operation.

10.2.2 Cache Lookup and Cache Design

As outlined in Section 10.2.1, we wish to design a 8 KB cache with a block size of 64 bytes for a 32-bit system. To do an efficient cache lookup, we need to find an efficient way to find out if the 26 bit block address exists among the 128 entries in the cache. There are thus two problems here. The first problem is to quickly locate a given entry, and the second is to perform a read/write operation. Instead of using a single SRAM array to solve both the problems, it is a better idea to split it into two arrays as shown in Figure 10.5.

In a typical design, a cache entry is saved in two SRAM based arrays. One SRAM array known as the *tag array*, contains information pertaining to the block address, and the other SRAM array known as the *data array* contains the data for the block. The tag array contains a *tag* that uniquely identifies a block. The tag is typically a part of the block address, and depends on the type of the cache. Along with the tag and data arrays, there is a dedicated cache controller that executes the cache access algorithm.

Fully Associative(FA) Cache

Let us first consider a very simple way of locating a block. We can check each of the 128 entries in the cache possibly simultaneously to see if the block address is equal to the block address in the cache entry. This cache is known as a fully associative cache or a content addressable cache. The phrase "fully associative" means that a given block can be *associated* with any entry in the cache.

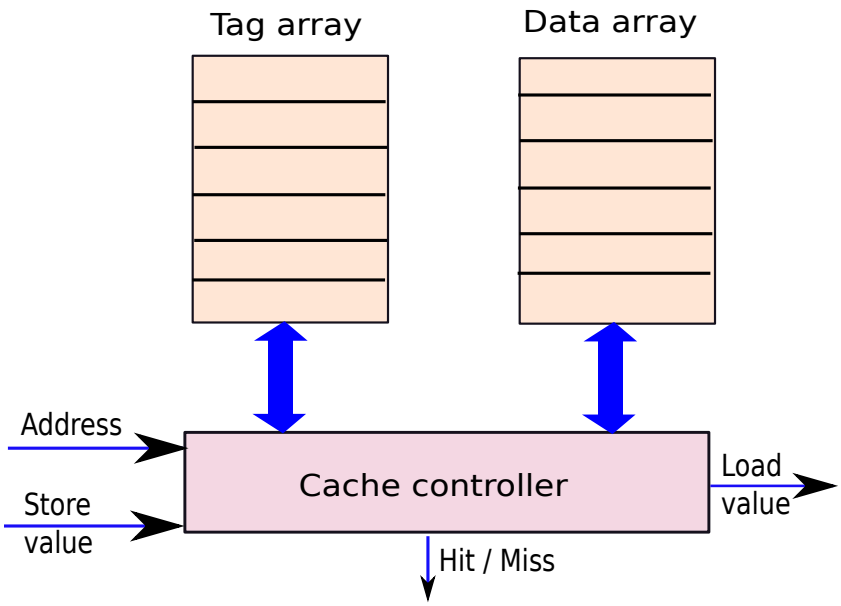


Figure 10.5: Structure of a cache

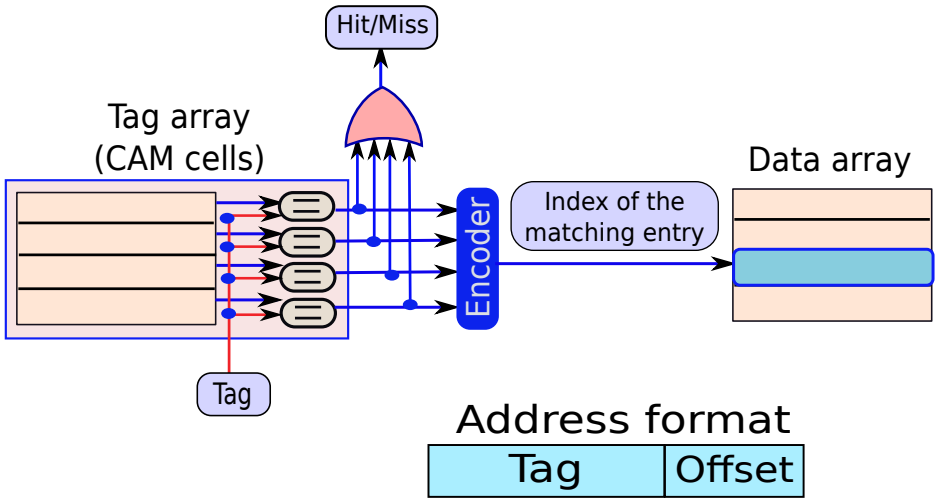


Figure 10.6: A fully associative cache

Each cache entry in a fully associative(FA) cache thus needs to contain two fields – *tag* and *data*. In this case, we can set the tag to be equal to the block address. Since the block address is unique to each block, it fits the definition of a *tag*. *Block data* refers to the contents of the block (64 bytes in this case). The block address requires 26 bits, and the block data requires 64 bytes, in our running example. The search operation needs to span the entire cache, and once

an entry is located, we need to either read out the data, or write a new value.

Let us first take a look at the tag array. Each tag in this case is equal to the 26 bit block address. After a memory request reaches a cache, the first step is to compute the tag by extracting the 26 most significant bits. Then, we need to match the extracted tag with each entry in the tag array using a set of comparators. If there is no match, then we can declare a cache miss and do further processing. However, if there is a cache hit, then we need to use the number of the entry that matches the tag to access the data entry. For example, in our 8 KB cache that contains 128 entries, it is possible that the 53^{rd} entry in the tag array matches the tag. In this case, the cache controller needs to fetch the 53^{rd} entry from the data array in the case of a read access, or write to the 53^{rd} entry in the case of a write access..

There are two ways to implement the tag array in a fully associative cache. Either we can design it as a normal SRAM array in which the cache controller iterates through each entry, and compares it with the given tag. Or, we can use a CAM array (see Section 6.4.2) that has comparators in every row. They can compare the value of the tag with the data stored in the row and produce an output (1 or 0) depending on the result of the comparison. A CAM array typically uses an encoder to compute the number of the row that matches the result. A CAM implementation of the tag array of a fully associative cache is more common, primarily because sequentially iterating through the array is very time consuming.

Figure 10.6 illustrates this concept. We enable each row of the CAM array by setting the corresponding word line to 1. Subsequently, the embedded comparators in the CAM cells compare the contents of each row with the tag, and generate an output. We use an OR gate to determine if any of the outputs is equal to 1. If any of the outputs is 1, then we have a cache hit, otherwise, we have a cache miss. Each of these output wires are also connected to an encoder that generates the index of the row that has a match. We use this index to access the data array and read the data for the block. In the case of a write, we write to the block, instead of reading it.

A fully associative cache is very useful for small structures (typically 2-32) entries. However, it is not possible to use CAM arrays for larger structures. The area and power overheads of comparison, and encoding are very high. It is also not possible to sequentially iterate through every entry of an SRAM implementation of the tag array. This is very time consuming. Hence, we need to find a better way for locating data in larger structures.

Direct Mapped(DM) Cache

We saw that in a fully associative cache, we can store any block at any location in the cache. This scheme is very flexible; however, it cannot be used when the cache has a large number of entries primarily because of prohibitive area and power overheads. Instead of allowing a block to be stored anywhere in the cache, let us assign only one fixed location for a given block. This can be done as follows.

In our running example, we have a 8 KB cache with 128 entries. Let us restrict the placement of 64 byte blocks in the cache. For each block, let us assign a unique location in the tag array at which the tag corresponding to its address can be stored. We can generate such a unique location as follows. Let us consider the address of a block A, and the number of entries in our cache (128), and compute $A\%128$. The % operator computes the remainder of the division of A by 128. Since A is a binary value, and 128 is a power of 2, computing the remainder is very

easy. We need to just extract the 7 LSB bits out of the 26-bit block address. These 7 bits can then be used to access the tag array. We can then compare the value of the tag saved in the tag array with the tag computed from the block address to determine if we have a hit or a miss.

Instead of saving the block address in the tag array as we did for a fully associative cache, we can slightly optimise its design. We observe that 7 out of the 26 bits in the block address are used to access the tag in the tag array. This means that all the blocks that can possibly be mapped to a given entry in the tag array will have their last 7 bits common. Hence, these 7 bits need not explicitly be saved as a part of the tag. We need to only save the remaining 19 bits of the block address that can vary across blocks. Thus a tag in a direct mapped implementation of our cache needs to contain 19 bits only.

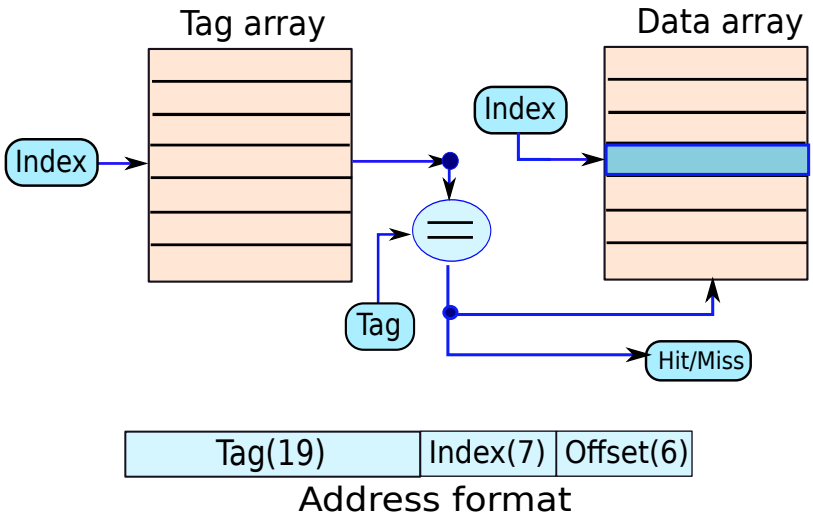


Figure 10.7: A direct mapped cache

Figure 10.7 describes this concept graphically. We divide a 32-bit address into three parts. The most significant 19 bits comprise the tag, the next 7 bits are referred to as the *index* (index in the tag array), and the remaining 6 bits point to the offset of the byte in the block. The rest of the access protocol is conceptually similar to that of a fully associative cache. In this case, we use the index to access the corresponding location in the tag array. We read the contents and compare it with the computed tag. If they are equal, then we declare a cache hit, otherwise, we declare a cache miss. Subsequently, in the case of a cache hit, we use the index to access the data array. In this case, we use the cache hit/miss result, to enable/disable the data array.

Way Point 10
Up till now we have taken a look at the fully associative and direct mapped caches.

- *The fully associative cache is a very flexible structure since a block can be saved in any entry in the cache. However, it has higher latency and power consumption. Since*

a given block can potentially be allocated in more entries of the cache, it has a higher hit rate than the direct mapped cache.

- *The direct mapped cache on the other hand is a faster and less power consuming structure. Here, a block can reside in only one entry in the cache. Thus, the expected hit rate of this cache is less than that of a fully associative cache.*

We thus observe that there is a tradeoff between power, latency, and hit rate between the fully associative and direct mapped caches.

Set Associative Cache

A fully associative cache is more power consuming because we need to search for a block in all the entries of the cache. In comparison, a direct mapped cache is faster and power efficient because we need to check just one entry. However, it clearly has a lower hit rate, and that is not acceptable either. Hence, let us try to combine both the paradigms.

Let us design a cache in which a block can potentially reside in any one of a set of multiple entries in a cache. Let us associate a *set* of entries in the cache with a block address. Like a fully associative cache, we will have to check all the entries in the set before declaring a hit or a miss. This approach combines the advantages of both the fully associative and direct mapped schemes. If a set contains 4 or 8 entries, then we do not have to use an expensive CAM structure, nor, do we have to sequentially iterate through all the entries. We can simply read out all the entries of the set from the tag array in parallel and compare all of them with the tag part of the block address in parallel. If there is a match, then we can read the corresponding entry from the data array. Since multiple blocks can be associated with a set, we call this design a *set associative* cache. The number of blocks in a set is known as the associativity of the cache. Secondly, each entry in a set is known as a *way*.

Definition 103

Associativity *The number of blocks contained in a set is defined as the* associativity *of the cache.*

Way *Each entry in a set is known as also known as a* way.

Let us now describe a simple method to group cache entries into sets for our simple example, in which we considered a 32-bit memory system with an 8-KB cache and 64-byte blocks. As shown in Figure 10.8, we first remove the lowest 6 bits from the 32-bit address because these specify the address of a byte within a block. The remaining 26 bits specify the block address. Our 8-KB cache has a total of 128 entries. If we want to create sets containing 4 entries each,

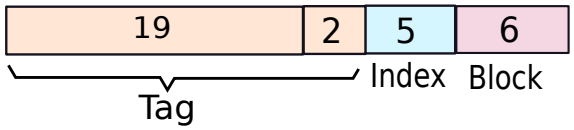


Figure 10.8: Division of a block address into sets

then we need to divide all the cache entries into sets of 4 entries. There will be $32(2^5)$ such sets.

In a direct mapped cache, we devoted the lowest 7 bits out of the 26 bit block address to specify the index of the entry in the cache. We can now split these 7 bits into two parts as shown in Figure 10.8. One part contains 5 bits and indicates the address of the set, and the second part containing 2 bits is ignored. The group of 5 bits indicating the address of the set is known as the *set index*.

After computing the set index, i, we need to access all the elements belonging to the set in the tag array. We can arrange the tag array as follows. If the number of blocks in a set is S, then we can group all the entries belonging to a set contiguously. For the i^{th} set, we need to access the elements iS, $(iS+1)$ ... $(iS+S-1)$ in the tag array.

For each entry in the tag array, we need to compare the tag saved in the entry to the *tag part* of the block address. If there is a match, then we can declare a hit. The notion of a tag in a set associative cache is rather tricky. As shown in Figure 10.8, it consists of the bits that are not a part of the index. In the case of our running example, it is the (21=26-5) MSB bits of the block address. The logic for deciding the number of tag bits is as follows.

Each set is specified by a 5-bit set index. These 5 bits are common to all the blocks that can be potentially mapped to the given set. We need to use the rest of the bits (26-5=21) to distinguish between the different blocks that are mapped to the same set. Thus, a tag in a set associative cache has a size between that of a direct mapped cache (19) and a fully associative cache (26).

Figure 10.9 shows the design of a set associative cache. We first compute the set index from the address of the block. For our running example, we use bits 7-11. Subsequently, we use the set index to generate the indices of its corresponding four entries in the tag array using the *tag array index generator*. Then, we access all the four entries in the tag array in parallel, and read their values. It is not necessary to use a CAM array here. We can use a single multi-port (multiple input, output) SRAM array. Next, we compare each element with the tag, and generate an output (0 or 1). If any of the outputs is equal to 1 (determined by an OR gate), then we have a cache hit. Otherwise, we have a cache miss. We use an encoder to find the index of the tag in the set that matched. Since, we are assuming a 4 way associative cache, the output of the encoder is between 00 to 11. Subsequently, we use a multiplexer to choose the index of the matching entry in the tag array. This index, can now be used to access the data array. The corresponding entry in the data array contains the data for the block. We can either read it or write to it.

We can perform a small optimisation here, for read operations. Note that in the case of a read operation, the access to the data array and tag array can proceed in parallel. If a set has

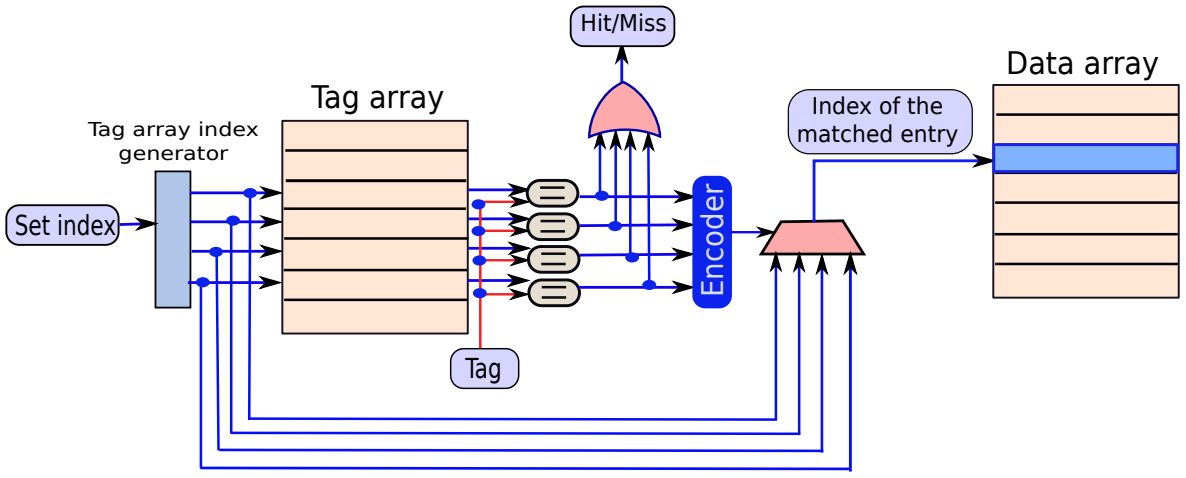


Figure 10.9: A set associative cache

4 ways, then while we are computing a tag match, we can read the 4 data blocks corresponding to the 4 ways of the set. Subsequently, in the case of a cache hit, and after we have computed the matching entry in the tag array, we can choose the right data block using a multiplexer. In this case, we are effectively overlapping some or all of the time required to read the blocks from the data array with the tag computation, tag array access, and match operations. We leave the resulting circuit as an exercise to the reader.

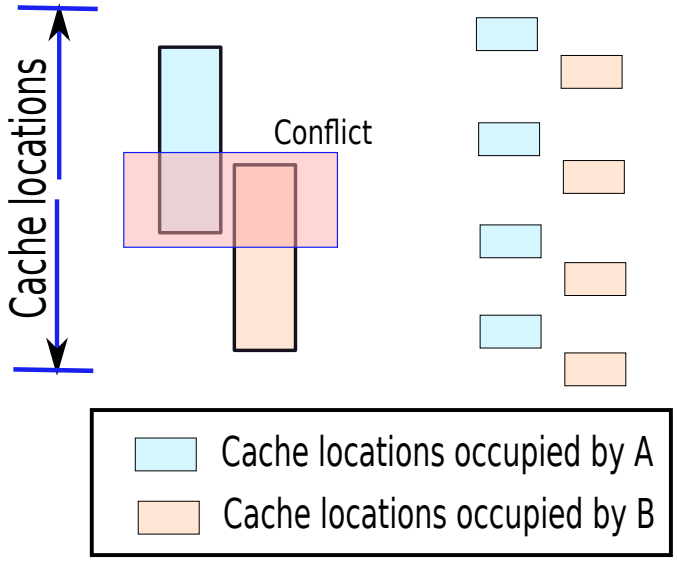
To conclude, we note that the set associative cache is by far the most common design for caches. It has acceptable power consumption values and latencies for even very large caches. The associativity of a set associative cache is typically 2, 4 or 8. A set with an associativity of K is also known as a $K - way$ associative cache.

Important Point 16 *We need to answer a profound question while designing a set associative cache. What should be the relative ordering of the set index bits and the ignored bits? Should the ignored bits be towards the left (MSB) of the index bits, or towards the right (LSB) of the index bits? In Figure 10.8, we have chosen the former option. What is the logic behind this?*

Answer: *If we have the ignored bits to the left(MSB) of the index bits, then contiguous blocks map to different sets. However, for the reverse case in which the ignored bits are to the right(LSB) of the index bits, contiguous blocks map to the same set. Let us call the former scheme* ***NON-CONT****, and the latter scheme* ***CONT****. We have chosen NON-CONT in our design.*

Let us consider two arrays, A, and B. Let the sizes of A and B be significantly smaller than the size of the cache. Moreover, let some of their constituent blocks map to the same group of sets. The figure below shows a conceptual map of the regions of the cache that store both the arrays for the CONT and NON-CONT schemes. We observe that even though, we have sufficient space in the cache, it is not possible to save both the arrays in the cache

concurrently using the CONT scheme. Their memory footprints overlap in a region of the cache, and it is not possible to save data for both the programs simultaneously in the cache. However, the NON-CONT scheme tries to uniformly distribute the blocks across all the sets. Thus, it is possible to save both the arrays in the cache at the same time.



This is a frequently occurring pattern in programs. The CONT scheme reserves an entire area of the cache and thus it is not possible to accommodate other data structures that map to conflicting sets. However, if we distribute the data in the cache, then we can accommodate many more data structures and reduce conflicts.

Example 120

A cache has the following parameters in a 32-bit system.

Parameter	*Value*
Size	N
Associativity	K
Block Size	B

What is the size of the tag?

Answer:

- *The number of bits required to specify a byte within a block is* $log(B)$.
- *The number of blocks is equal to* N/B*, and the number of sets is equal to* $N/(BK)$.
- *Thus, the number of set index bits is equal to:* $log(N) - log(B) - log(K)$.

- *The remaining number of bits are tag bits. It is equal to:* $32 - (log(N) - log(B) - log(K) + log(B)) = 32 - log(N) + log(K)$.

10.2.3 *Data read* and *data write* Operations

The *data read* Operation

Once, we have established that a given block is present in a cache, we use the *basic* read operation to get the value of the memory location from the data array. We establish the presence of a block in a cache if the *lookup* operation returns a cache hit. If there is a miss in the cache, then the cache controller needs to raise a read request to the lower level cache, and fetch the block. The *data read* operation can start as soon as data is available.

The first step is to read out the block in the data array that corresponds to the matched tag entry. Then, we need to choose the appropriate set of bytes out of all the bytes in the block. We can use a set of multiplexers to achieve this. The exact details of the circuit are left to the reader as an exercise.

Secondly, as described in Section 10.2.2, it is not strictly necessary to start the *data read* operation after the *lookup* operation. We can have a significant overlap between the operations. For example, we can read the tag array and data array in parallel. We can subsequently select the right set of values using multiplexers after the matching tag has been computed.

The *data write* Operation

Before, we can write a value, **we need to ensure that the entire block is already present in the cache**. This is a very important concept. Note that we cannot make an argument that since we are creating new data, we do not need the previous value of the block. The reason is as follows. We typically write 4 bytes or at the most 8 bytes for a single memory access. However, a block is at least 32 or 64 bytes long. A *block* is an atomic unit in our cache. Hence, we cannot have different parts of it at different places. For example, we cannot save 4 bytes of a block in the L1 cache, and the rest of the bytes in the L2 cache. Secondly, for doing so, we need to maintain additional state that keeps track of the bytes that have been updated with writes. Hence, in the interest of simplicity, even if we wish to write just 1 byte, we need to populate the cache with the entire block.

After that we need to write the new values in the data array by enabling the appropriate set of word lines and bit lines. We can design a simple circuit to achieve this using a set of demultiplexers. The details are left to the reader.

There are two methods of performing a data write – write-back and write-through. Write-through is a relatively simpler scheme. In this approach, whenever we write a value into the data array, we also send a write operation to the lower level cache. This approach increases the amount of cache traffic. However, it is simpler to implement the cache because we do not have to keep track of the blocks that have been modified after they were brought into the cache. We can thus seamlessly evict a line from the cache if required. Here cache evictions and replacements are simple, at the cost of writes. We shall also see in Chapter 11 that it is easy to implement caches for mutiprocessors if the L1 caches follow a write-through protocol.

In the write-back scheme, we explicitly keep track of blocks that have been modified using write operations. We can maintain this information by using an additional bit in the tag array. This bit is typically known as the *modified* bit. Whenever, we get a block from the lower level of the memory hierarchy, the modified bit is 0. However, when we do a *data write* and update the data array, we set the modified bit in the tag array to 1. Evicting a line requires us to do extra processing that we shall describe in Section 10.2.6. For a write-back protocol, writes are cheap, and evict operations are more expensive. The tradeoff here is the reverse of that in write-through caches.

The structure of an entry in the tag array with the additional modified bit is shown in Figure 10.10.

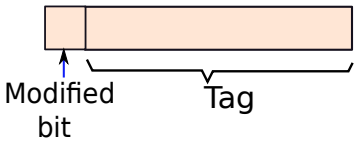


Figure 10.10: An entry in the tag array with the modified bit

10.2.4 The *insert* Operation

In this section, we shall discuss the protocol to insert a block in a cache. This operation is invoked when a block arrives from a lower level. We need to first take a look at all the ways of the set that a given block is mapped to, and see if there are any empty entries. If there are empty entries then we can choose one of the entries arbitrarily, and populate it with the contents of the given block. If we do not find any empty entries, we need to invoke the *replace* and *evict* operations to choose and remove an already existing block from the set.

We need to maintain some extra status information to figure out if a given entry is empty or non-empty. In computer architecture parlance, these states are also known as *invalid* and *valid* respectively. We need to store just 1 extra bit in the tag array to indicate the status of a block. It is known as the *valid* bit. We shall use the tag array for saving additional information regarding an entry, because it is smaller and typically faster than the data array.

The structure of an entry in the tag array with the addition of the valid bit is shown in Figure 10.11.

The cache controller needs to check the valid bits of each of the tags while searching for invalid entries. Note that all the entries of a cache are invalid initially. If an invalid entry is found, then the corresponding entry in the data array can be populated with the contents of the block. The entry subsequently becomes valid. However, if there is no invalid entry, then we need to replace one entry with the given block that needs to be inserted into the cache.

10.2.5 The *replace* Operation

The task is here to find an entry in the set that can be replaced by a new entry. We do not wish to replace an element that is accessed very frequently. This will increase the number of cache misses. We ideally want to replace an element that has the least probability of being

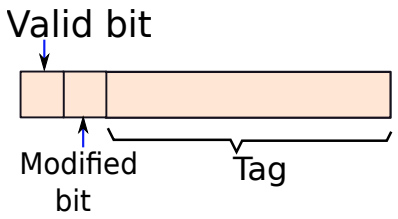


Figure 10.11: An entry in the tag array with the modified, and valid bits

accessed in the future. However, it is difficult to predict future events. Hence, we need to make reasonable guesses based on past behavior. We can have different policies for the replacement of blocks in a cache. These are known as *replacement schemes* or *replacement policies*.

Definition 104
A cache replacement scheme *or* replacement policy *is a method to replace an entry in the set by a new entry.*

Random Replacement Policy

The most trivial replacement policy is known as the *random* replacement policy. Here, we pick a block at random and replace it. This scheme is very simple to implement. However, it is not very optimal in terms of performance, because it does not take into account the behaviour of the program and the nature of the memory access pattern. This scheme ends up often replacing very frequently accessed blocks.

FIFO Replacement Policy

The next scheme is slightly more complicated, and is known as the *FIFO (first in first out)* replacement policy. Here, the assumption is that the block that was brought into the cache at the earliest point of time, is the least likely to be accessed in the future. To implement the FIFO replacement policy, we need to add a counter to the tag array. Whenever, we bring in a block, we assign it a counter value equal to 0. We increment the counter values for the rest of the blocks. The larger is the counter, the earlier the block was brought into the cache.

Now to find a candidate for replacement, we need to find an entry with the largest value of the counter. This must be the earliest block. Unfortunately, the FIFO scheme does not strictly align with our principles of temporal locality. It penalises blocks that are present in the cache for a long time. However, they may also be very frequently accessed blocks, and should not be evicted in the first place.

Let us now consider the practical aspects of implementing a FIFO replacement policy. The maximum size of the counter needs to be equal to the number of elements in a set, i.e., the

associativity of the cache. For example, if the associativity of a cache is 8, we need to have a 3 bit counter. The entry that needs to be replaced should have the largest counter value.

Note that in this case, the process of bringing in a new value into the cache is rather expensive. We need to increment the counters of all the elements in the set except one. However, cache misses, are more infrequent as compared to cache hits. Hence, the overhead is not significant in practice, and this scheme can be implemented without large performance overheads.

LRU Replacement Policy

The LRU (least recently used) replacement policy is known to be as one of the most efficient schemes. The LRU scheme follows directly from the definition of stack distance. We ideally want to replace a block that has the lowest chance of being accessed in the future. According to the notion of stack distance, the probability of being accessed in the future is related to the probability of accesses in the recent past. If a processor has been accessing a block frequently in the last window of n (n is not a very large number) accesses, then there is a high probability that the block will be accessed in the immediate future. However, if the last time that a block was accessed is long back in the past, then the chances are unlikely that it will be accessed soon.

In the LRU replacement policy, we maintain the time that a block was last accessed. We choose the block that was last accessed at the earliest point of time as a candidate for replacement. In a hypothetical implementation of a LRU replacement policy, we maintain a timestamp for every block. Any time that a block is accessed, its timestamp is updated to match the current time. For finding an appropriate candidate for replacement, we need to find the entry with the smallest timestamp in a set.

Let us now consider the implementation of an LRU scheme. The biggest issue is that we need to do additional work for every read and write access to the cache. There will be a significant performance impact because typically 1 in 3 instructions are memory accesses. Secondly, we need to dedicate bits to save a timestamp that is sufficiently large. Otherwise, we need to frequently reset the timestamps of every block in a set. This process will induce a further slowdown, and additional complexity in the cache controller. Implementing an LRU scheme that is as close to an ideal LRU implementation as possible, and that does not have significant overheads, is thus a difficult task.

Hence, let us try to design LRU schemes that use small timestamps (typically 1-3 bits), and approximately follow the LRU policy. Such kind of schemes are called pseudo-LRU schemes. Let us outline a simple method for implementing a basic pseudo-LRU scheme. Note that we can have many such approaches, and the reader is invited to try different approaches and test them on a cache simulator such as Dinero [Edler and Hill, 1999], or sim-cache [Austin et al., 2002]. Instead of trying to explicitly mark the least recently used element, let us try to mark the more recently used elements. The elements that are not marked will automatically get classified as the least recently used elements.

Let us start out by associating a counter with each block in the tag array. Whenever, a block is accessed (read/write), we increment the counter. However, once the counter reaches the maximum value, we stop incrementing it further. For example, if we use a 2-bit counter, then we stop incrementing the counter beyond 3. Now, we need to do something more. Otherwise, the counter associated with every block will ultimately reach 3 and stay there. To solve this

problem, we can periodically decrement the counters of every block in a set by 1, or we can even reset them to 0. Subsequently, some of the counters will start increasing again. This procedure will ensure that for most of the time, we can identify the least recently used blocks by taking a look at the value of counters. The block associated with the lowest value of the counter is one of the least recently used blocks, and most likely "the most least recently used block". Note that this approach does involve some amount of activity per access. However, incrementing a small counter has little additional overhead. Secondly, it is not in the critical path in terms of timing. It can be done in parallel or sometime later also. Finding a candidate for replacement involves looking at all the counters in a set, and finding the block with the lowest value of the counter. After we replace the block with a new block, most processors typically set the counter of the new block to the largest possible value. This indicates to the cache controller, that the new block should have the least priority with respect to being a candidate for replacement.

10.2.6 The *evict* Operation

Lastly, let us take a look at the *evict* operation. If the cache follows a write-through policy, then nothing much needs to be done. The block can simply be discarded. However, if the cache follows a write-back policy, then we need to take a look at the modified bit. If the data is not modified, then it can be seamlessly evicted. However, if the data has been modified, then it needs to written back to the lower level cache.

10.2.7 Putting all the Pieces Together

Cache Read Operation

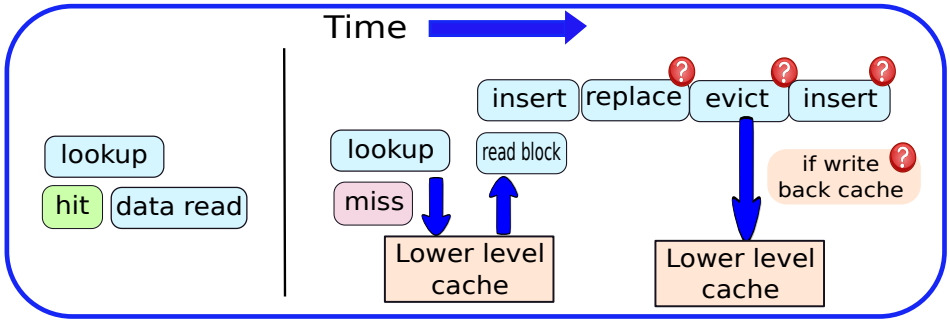


Figure 10.12: The *read* operation

The sequence of steps in a cache read operation is shown in Figure 10.12. We start with a *lookup* operation. As mentioned in Section 10.2.2, we can have a partial overlap between the *lookup* and *data read* operations. If there is a cache hit, then the cache returns the value to the processor, or the higher level cache (whichever might be the case). However, if there is a cache miss, then we need to cancel the *data read* operation, and send a request to the lower level cache. The lower level cache will perform the same sequence of accesses, and return the entire cache block (not just 4 bytes). The cache controller can then extract the requested data from

the block, and send it to the processor. Simultaneously, the cache controller invokes the *insert* operation to insert the block into the cache. If there is an *invalid* entry in the set, then we can replace it with the given block. However, if all the ways in a set are valid, it is necessary to invoke the *replace* operation to find a candidate for replacement. The figure appends a question mark with this operation, because this operation is not invoked all the time (only when all the ways of a set contain valid data). Then, we need to *evict* the block, and possibly write it to the lower level cache if the line is modified, and we are using a write-back cache. The cache controller then invokes the *insert* operation. This time it is guaranteed to be successful.

Cache Write Operation (write-back Cache)

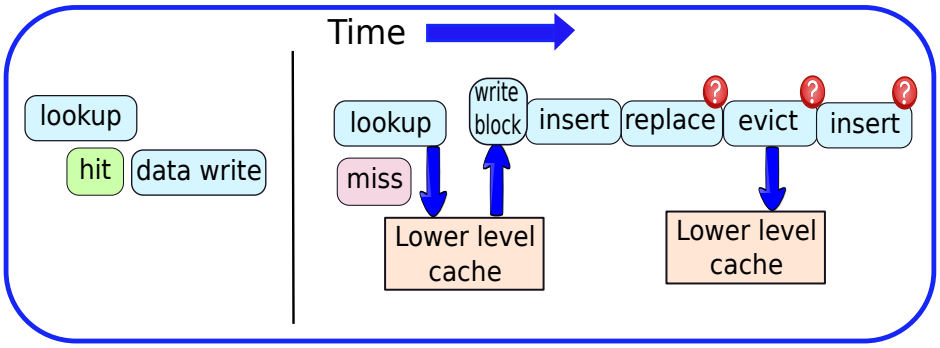


Figure 10.13: The *write* operation (write-back Cache)

Figure 10.13 shows the sequence of operations for a cache write operation for a write-back cache. The sequence of operations are roughly similar to that of a cache read. If there is a cache hit, then we invoke a *data write* operation, and set the modified bit to 1. Otherwise, we issue a read request for the block to the lower level cache. After the block arrives, most cache controllers typically store it in a small temporary buffer. At this point, we write the 4 bytes (that we are interested in) to the buffer, and return. In some processors, the cache controller might wait till all the sub-operations complete. After writing into the temporary buffer (*write block* operation in Figure 10.13), we invoke the *insert* operation for writing the contents (modified) of the block. If this operation is not successful (because all the ways are valid), then we follow the same sequence of steps as the *read operation* (*replace*, *evict*, and *insert*).

Cache Write Operation (write-through Cache)

Figure 10.14 shows the sequence of operations for a write-through cache. The first point of difference is that we write the block to the lower level, even if the request hits in the cache. The second point of difference is that after we write the value into the temporary buffer (after a miss), we write back the new contents of the block to the lower level cache also. The rest of the steps are similar to the sequence of steps that we followed for the write-back cache.

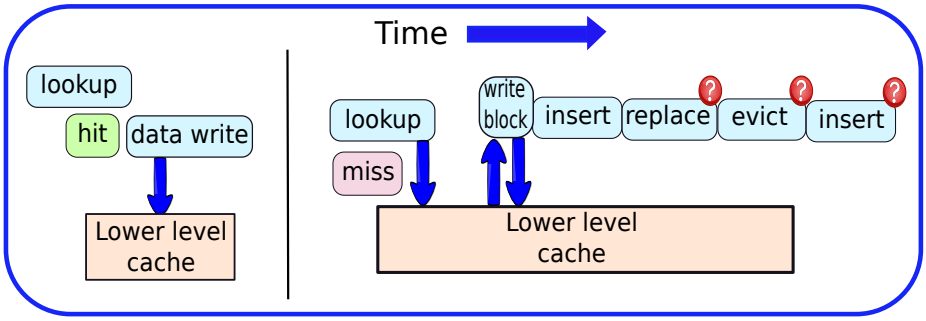


Figure 10.14: The *write* operation (write-through Cache)

10.3 The Memory System

We now have a fair understanding of the working of a cache, and all its constituent operations. A memory system is built using a hierarchy of caches as mentioned in Section 10.1.7. The memory system as a whole supports two basic operations: *read*, and *write*, or alternatively, *load* and *store*.

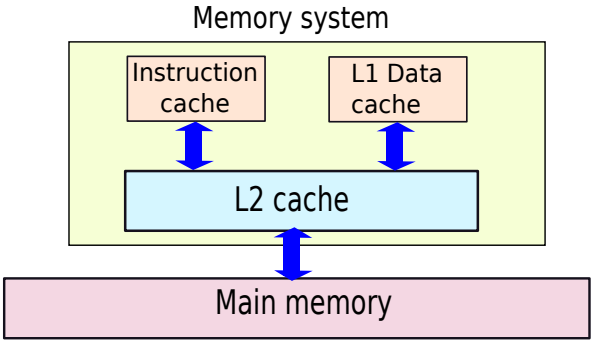


Figure 10.15: The memory system

We have two caches at the highest level – The data cache (also referred to as the L1 cache), and the instruction cache (also referred to as the I Cache). Almost all the time both of them contain different sets of memory locations. The protocol for accessing the I Cache and L1 cache is the same. Hence, to avoid repetition let us just focus on the L1 cache from now on. The reader needs to just remember that accesses to the instruction cache follow the same sequence of steps.

The processor starts by accessing the L1 cache. If there is a L1 hit, then it typically receives the value within 1-2 cycles. Otherwise, the request needs to go to the L2 cache, or possibly even lower levels such as the main memory. In this case, the request can take tens or hundreds of cycles. In this section, we shall look at the system of caches in totality, and treat them as one single black box referred to as the *memory system*.

If we consider inclusive caches, which is the convention in most commercial systems, the total size of the memory system is equal to the size of the main memory. For example, if a system has 1 GB of main memory, then the size of the memory system is equal to 1 GB. It is possible that internally, the memory system might have a hierarchy of caches for improving performance. However, they do not add to the total storage capacity, because they only contain subsets of the data contained in main memory. Moreover, the memory access logic of the processor also views the entire memory system as a single unit, conceptually modelled as a large array of bytes. This is also known as the physical memory system, or the *physical address space*.

Definition 105
The physical address space comprises of the set of all memory locations contained in the caches, and main memory.

10.3.1 Mathematical Model of the Memory System

Performance

The memory system can be thought of as a black box that just services read and write requests. The time a request takes is variable. It depends on the level of the memory system at which the request hits. The pipeline is attached to the memory system in the memory access (MA) stage, and issues requests to it. If the reply does not come within a single cycle, then additional pipeline bubbles need to be introduced in our 5 stage *SimpleRisc* in-order pipeline.

Let the average memory access time be $AMAT$ (measured in cycles), and the fraction of load/store instructions be f_{mem}. Then the CPI can be expressed as:

$$\begin{aligned} CPI &= CPI_{ideal} + stall\_rate * stall\_cycles \\ &= CPI_{ideal} + f_{mem} \times (AMAT - 1) \end{aligned} \tag{10.1}$$

CPI_{ideal} is the CPI assuming a perfect memory system having a 1 cycle latency for all accesses. Note that in our 5 stage in-order pipeline the ideal instruction throughput is 1 instruction per cycle, and the memory stage is allotted 1 cycle. In practice, if a memory access takes n cycles, then we have $n - 1$ stall cycles, and they need to be accounted for by Equation 10.1. In this equation, we implicitly assume that every memory access suffers a stall for $AMAT - 1$ cycles. In practice this is not the case since most of the instructions will hit in the L1 cache, and the L1 cache typically has a 1 cycle latency. Hence, accesses that hit in the L1 cache will not stall. However, long stall cycles will be introduced by accesses that miss in the L1 and L2 caches.

Nonetheless, Equation 10.1 still holds because we are only interested in the average CPI for a large number of instructions. We can derive this equation by considering a large number of instructions, summing up all the memory stall cycles, and computing the average number of cycles per instruction.

Average Memory Access Time

In Equation 10.1, CPI_{ideal} is determined by the nature of the program and the nature of the other stages (other than MA) of the pipeline. f_{mem} is also an inherent property of the program running on the processor. We need a formula to compute $AMAT$. We can compute it in a way similar to Equation 10.1.

Assuming, a memory system with an L1 and L2 cache, we have:

$$\begin{aligned} AMAT &= L1_{hit\,time} + L1_{miss\,rate} \times L1_{miss\,penalty} \\ &= L1_{hit\,time} + L1_{miss\,rate} \times (L2_{hit\,time} + L2_{miss\,rate} \times L2_{miss\,penalty}) \end{aligned} \tag{10.2}$$

All the memory accesses need to access the L1 cache irrespective of a hit or a miss. Hence, they need to incur a delay equal to $L1_{hit\,time}$. A fraction of accesses, $L1_{miss\,rate}$, will miss in the L1 cache, and move to the L2 cache. Here also, irrespective of a hit or a miss, we need to incur a delay of $L2_{hit\,time}$ cycles. If a fraction of accesses ($L2_{miss\,rate}$) miss in the L2 cache, then they need to proceed to main memory. We have assumed that all the accesses hit in the main memory. Hence, the $L2_{miss\,penalty}$ is equal to the main memory access time.

Now, if we assume that we have a n level memory system where the first level is the L1 cache, and the last level is the main memory, then we can use a similar equation.

$$\begin{aligned} AMAT &= L1_{hit\,time} + L1_{miss\,rate} \times L1_{miss\,penalty} \\ L1_{miss\,penalty} &= L2_{hit\,time} + L2_{miss\,rate} \times L2_{miss\,penalty} \\ L2_{miss\,penalty} &= L3_{hit\,time} + L3_{miss\,rate} \times L3_{miss\,penalty} \\ \ldots &= \ldots \\ L(n-1)_{miss\,penalty} &= Ln_{hit\,time} \end{aligned} \tag{10.3}$$

We need to note that the miss rate used in these equations for a certain level i is equal to the number of accesses that miss at that level divided by the total number of accesses to that level. This is known as the *local miss rate*. In comparison, we can define a *global miss rate* for level i, which is equal to the number of misses at level i divided by the total number of memory accesses.

Definition 106

local miss rate *It is equal to the number of misses in a cache at level i divided by the total number of accesses at level i.*

global miss rate *It is equal to the number of misses in a cache at level i divided by the total number of memory accesses.*

Let us take a deeper look at Equation 10.1. We observe that we can increase the performance of a system by either reducing the miss rate, the miss penalty or by decreasing the hit time. Let us first look at the miss rate.

10.3.2 Cache Misses

Classification of Cache Misses

Let us first try to categorise the different kinds of misses in a cache.

The first category of misses are known as *compulsory* misses or cold misses. These misses happen, when data is loaded into a cache for the first time. Since the data values are not there in the cache, a miss is bound to happen. The second category of cache misses are known as *capacity* misses. We have a capacity miss, when the amount of memory required by a program is more than the size of the cache. For example, let us assume that a program repeatedly accesses all the elements of an array. The size of the array is equal to 1 MB, and the size of the L2 cache is 512 KB. In this case, there will be capacity misses in the L2 cache, because it is too small to hold all the data. The set of blocks that a program accesses in a typical interval of time is known as its *working set*. We can thus alternatively say that conflict misses happen when the size of the cache is smaller than the working set of the program. Note that the definition of the working set is slightly imprecise because the length of the interval is considered rather subjectively. However, the connotation of the time interval is that it is a small interval compared to the total time of execution of the program. Nevertheless, it is large enough to ensure that the behaviour of the system achieves a steady state. The last category of misses are known as *conflict* misses. These misses occur in direct mapped and set associative caches. Let us consider a 4 way set associative cache. If there are 5 blocks that map to the same set in the working set of a program, then we are bound to have cache misses. This is because the number of blocks accessed is larger than the maximum number of entries that can be part of a set. These misses are known as conflict misses.

Definition 107
The memory locations accessed by a program in a short interval of time comprise the working set of the program at that point of time.

The categorisation of misses into these three categories – compulsory, capacity, and conflict – is also known as the three 'C's.

Reduction of the Miss Rate

To sustain a high IPC, it is necessary to reduce the cache miss rate. We need to adopt different strategies to reduce the different kinds of cache misses.

Let us start out with compulsory misses. We need a method to predict the blocks that will be accessed in the future, and fetch the blocks in advance. Typically schemes that leverage spatial locality serve as effective predictors. Hence, increasing the block size should prove beneficial in reducing the number of compulsory misses. However, increasing the block size beyond a certain limit can have negative consequences also. It reduces the number of blocks that can be saved in a cache, and secondly the additional benefit might be marginal. Lastly, it will take more time to read and transfer bigger blocks from the lower levels of the memory system. Hence, designers avoid very large block sizes. Any value between 32-128 bytes is reasonable.

Modern processors typically have sophisticated predictors that try to predict the addresses of blocks that might be accessed in the future based on the current access pattern. They subsequently fetch the predicted blocks from the lower levels of the memory hierarchy in an attempt to reduce the miss rate. For example, if we are sequentially accessing the elements of a large array, then it is possible to predict the future accesses based on the access pattern. Sometimes we access elements in an array, where the indices differ by a fixed value. For example, we might have an algorithm that accesses every fourth element in an array. In this case also, it is possible to analyse the pattern and predict future accesses because the addresses of consecutive accesses differ by the same value. Such kind of a unit is known as a *hardware prefetcher*. It is present in most modern processors, and uses sophisticated algorithms to "prefetch" blocks and consequently reduce the miss rate. Note that the hardware prefetcher should not be very aggressive. Otherwise, it will tend to displace more useful data from the cache than it brings in.

Definition 108
A hardware prefetcher is a dedicated hardware unit that predicts the memory accesses in the near future, and fetches them from the lower levels of the memory system.

Let us now consider capacity misses. The only effective solution is to increase the size of the cache. Unfortunately, the cache design that we have presented in this book requires the size of the cache to be equal to a power of two (in bytes). It is possible to violate this rule by using some advanced techniques. However, by and large most of the caches in commercial processors have a size that is a power of two. Hence, increasing the size of a cache is tantamount to at least doubling its size. Doubling the size of a cache requires twice the area, slows it down, and increases the power consumption. Here again, prefetching can help if used intelligently and judiciously.

The classical solution to reduce the number of conflict misses is to increase the associativity of a cache. However, increasing the associativity of a cache increases the latency and power consumption of the cache also. Consequently, it is necessary for designers to carefully balance the additional hit rate of a set associative cache, with the additional latency. Sometimes, it is the case that there are conflict misses in a few sets in the cache. In this case, we can have a small fully associative cache known as the *victim cache* along with the main cache. Any block that is displaced from the main cache, can be written to the victim cache. The cache controller needs to first check the main cache, and if there is a miss, then it needs to check the victim cache, before proceeding to the lower level. A victim cache at level i can thus filter out some of the requests that go to level $(i+1)$.

Note that along with hardware techniques, it is possible to write programs in a "cache friendly" way. These methods can maximise temporal and spatial locality. It is also possible for the compiler to optimise the code for a given memory system. Secondly, the compiler can insert prefetching code such that blocks can be prefetched into the cache before they are actually used. Discussion of such techniques are beyond the scope of this book.

Let us now quickly mention two rules of thumb. Note that these rules are found to approximately hold empirically, and are by no means fully theoretically justified. The first is known

as the *Square Root Rule* [Hartstein et al., 2006]. It says that the miss rate is proportional to the square root of the cache size.

$$miss\,rate \propto \frac{1}{\sqrt{cache\,size}} \qquad \text{[Square Root Rule]} \tag{10.4}$$

Hartstein et. al. [Hartstein et al., 2006] try to find a theoretical justification for this rule, and explain the basis of this rule by using results from probability theory. From their experimental results, they arrive at a generic version of this rule that says that the exponent of the cache size in the Square Root Rule varies from -0.3 to -0.7.

The other rule is known as the "Associativity Rule". It states that the effect of doubling associativity is almost the same as doubling the cache size with the original associativity. For example, the miss rate of a 64 KB 4-way associative cache is almost the same as that of a 128 KB 2-way associative cache.

We would further like to caution the reader that the Associativity Rule and the Square Root Rule are just thumb rules, and do not hold exactly. They can be used as mere conceptual aids. We can always construct examples that violate these rules.

10.3.3 Reduction of Hit Time and Miss Penalty

Hit Time

The average memory access time can also be reduced by reducing the hit time and the miss penalty. To reduce the hit time, we need to use small and simple caches. However, by doing so, we increase the miss rate also.

Miss Penalty

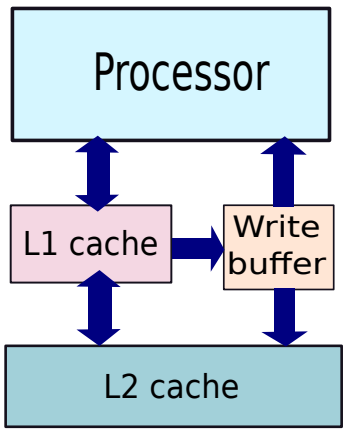


Figure 10.16: Write buffer

Let us now discuss ways to reduce the miss penalty. Note that the miss penalty at level i, is equal to the memory latency of the memory system starting at level $(i+1)$. The traditional

methods for reducing hit time, and miss rate can always be used to reduce the miss penalty at a given level. However, we are looking at methods that are exclusively targeted towards reducing the miss penalty. Let us first look at write misses in the L1 cache. In this case the entire block has to be brought into the cache from the L2 cache. This takes time (> 10 cycles), and secondly unless the write has completed, the pipeline cannot resume. Hence, processor designers use a small set associative cache known as a *write buffer* as shown in Figure 10.16. The processor can write the value to the write buffer, and then resume, or alternatively, it can write to the write buffer only if there is a miss in the L1 cache (as we have assumed). Any subsequent read needs to check the write buffer along with accessing the L1 cache. This structure is typically very small and fast (4-8 entries). Once, the data arrives in the L1 cache, the corresponding entry can be removed from the write buffer. Note that if a free entry is not available in the write buffer, then the pipeline needs to stall. Secondly, before the write miss has been serviced from the lower levels of the cache, it is possible that there might be another write to the same address. This can be seamlessly handled by writing to the allocated entry for the given address in the write buffer.

Let us now take a look at read misses. Let us start out by observing that the processor is typically interested in only up to 4 bytes per memory access. The pipeline can resume if it is provided those crucial 4 bytes. However, the memory system needs to fill the entire block before the operation can complete. The size of a block is typically between 32-128 bytes. It is thus possible to introduce an optimisation here, if the memory system is aware of the exact set of bytes that the processor requires. In this case, the memory system can first fetch the memory word (4 bytes) that is required. Subsequently, or in parallel it can fetch the rest of the block. This optimisation is known as *critical word first*. Then, this data can be quickly sent to the pipeline such that it can resume its operation. This optimisation is known as *early restart*. Implementing both of these optimisations increases the complexity of the memory system. However, *critical word first* and *early restart* are fairly effective in reducing the miss penalty.

10.3.4 Summary of Memory System Optimisation Techniques

Table 10.2 shows a summary of the different techniques that we have introduced to optimise the memory system. Note that every technique has some negative side effects. If a technique improves the memory system in one aspect, then it is detrimental in some other aspect. For example, by increasing the cache size we reduce the number of capacity misses. However, we also increase the area, latency, and power.

To summarise, we can conclude that it is necessary to design the memory system very carefully. The requirements of the target workload have to be carefully balanced with the constraints placed by the designers, and the limits of manufacturing technology. We need to maximise performance, and at the same time be mindful of power, area, and complexity constraints.

10.4 Virtual Memory

Up till now, we have considered only one program in our system. We have designed our entire system using this assumption. However, this assumption is not correct. For example,

Technique	Application	Disadvantages
large block size	compulsory misses	reduces the number of blocks in the cache
prefetching	compulsory misses, capacity misses	extra complexity and the risk of displacing useful data from the cache
large cache size	capacity misses	high latency, high power, more area
increased associativity	conflict misses	high latency, high power
victim cache	conflict misses	extra complexity
compiler based techniques	all types of misses	not very generic
small and simple cache	hit time	high miss rate
write buffer	miss penalty	extra complexity
critical word first	miss penalty	extra complexity and state
early restart	miss penalty	extra complexity

Table 10.2: Summary of different memory system optimisation techniques

at the moment there are 232 programs running on your author's workstation. The reader can easily find out the number of programs running on her system by opening the Task Manger on Windows, or by entering the command "ps -ef" on a Linux or a Macintosh system. It is possible for one processor to run multiple programs by switching between different programs very quickly. For example, while a user is playing a game, her processor might be fetching her new email. The reason she does not feel any interruption, is because the time scale at which the processor switches back and forth between programs (typically several milliseconds) is much smaller than what humans can perceive.

Secondly, we have assumed up till now that all the data that a program needs is resident in main memory. However, this assumption is also not correct. Back in the old days, the size of main memory used to be several megabytes, whereas, users could run very large programs that needed hundreds of megabytes of data. Even now, it is possible to work with data that is much larger than the amount of main memory. Readers can easily verify this statement, by writing a C program that creates data structures that are larger than the amount of physical memory contained in their machine. In most systems, this C program will compile and run successfully.

We shall see in this section that by making a small change in the memory system, we can satisfy both of these requirements.

10.4.1 Process – A Running Instance of a Program

Up till now, we have assumed the existence of only one program in the system. We assumed that it was in complete control of the memory system, and the processor pipeline. However, this is not the case in practice.

Let us first start out by accurately defining the notion of a *process* and differentiating it from a *program*. Up till now we have been loosely using the term – *program* – and sometimes using

it in place of a *process*. A program is an array of bytes and is saved as a file in the file system. The file is typically known as a *binary* or as an *executable*. The executable contains some meta data about the program such that its name and type, the constants used by the program, and the set of instructions. In comparison, a process is a running instance of a program. If we run one program several times, we create multiple processes. A process has access to the processor, peripheral devices, and the memory system. There is a dedicated area in the memory system that contains the data and code of the process. The program counter of the processor points to a given location in the code region of the process in memory when the process is executing. Memory values required by the process are obtained from its data region in the memory system. The *operating system* starts and ends a process, and manages it throughout its lifetime.

Definition 109
A process is a running instance of a program.

Operating System

Most of our readers must have heard of the term *operating system*. Most people mostly view an operating system such as Windows, Linux, or Mac OS X from the point of view of its user interface. However, this is a minor aspect of the operating system. It does many more things invisibly. Let us look at some of its important functionalities.

The operating system consists of a set of dedicated programs that manage the machine, peripheral devices, and all the processes running on the machine. Furthermore, the operating system facilitates efficient transfer of information between the hardware and software components of a computer system. The core component of an operating system is known as the *kernel*. Its main role is to manage the execution of processes, and manage memory. We shall look at the memory management aspect in Section 10.4.5. Let us now look at the process management aspect.

To run a program, a user needs to compile the program, and then either double click the program, or write the name of the program in the command line, and click the "enter" button. Once, this is done, the control passes to the operating system kernel. A component of the kernel known as the *loader* reads the content of the program, and copies it to a region in the memory system. Notably, it copies all the instructions in the *text* section, allocates space for all the data, and initialises memory with all the constants that a program will require during its execution. Subsequently, it initialises the values of registers, copies command line arguments to the stack, possibly initialises the stack pointer, and jumps to the entry point of the program. The user program can then begin to execute, in the context of a running process. Every process has a unique number associated with it. It is known as the *pid* (process id). After completion, it is the kernel's job to tear down the process, and reclaim all of its memory.

The other important aspect of process management is *scheduling*. A dedicated component of the *kernel* manages all the processes, including the kernel itself, which is a special process. It typically runs each process for a certain amount of time, and then switches to another process. As a user, we typically do not perceive this because every second, the kernel switches between processes hundreds of times. The time interval is too small for us to detect. However,

behind the scenes, the kernel is busy at work. For example, it might be running a game for sometime, running a program to fetch data from the network for some time, and then running some of its own tasks for sometime. The kernel also manages aspects of the file system, inter-process communication, and security. The discussion of such topics is beyond the scope of this book. The reader is referred to textbooks on operating systems such as the book by Tanenbaum [Tanenbaum, 2007] or Silbserchatz and Galvin [Silberschatz et al., 2008].

The other important components in an operating system are device drivers, and system utilities. *Device drivers* are dedicated programs that communicate with dedicated devices and ensure the seamless flow of information between them and user processes. For example, a printer and scanner have dedicated device drivers that make it possible to print and scan documents, respectively. Network interfaces have dedicated device drivers that allow us to exchange messages over the internet. Lastly, system utilities provide generic services to all the processes such as file management, device management (Control Panel in Windows), and security.

Definition 110

Operating System *The operating system consists of a set of dedicated programs that manage the machine, peripheral devices, and the processes running on it. It facilitates the transfer of information between the hardware and software components of a computer system.*

Kernel *The kernel is a program that is the core of the operating system. It has complete control over the rest of the processes in the operating system, the user processes, the processor, and all external devices. It mainly performs the task of managing multiple processes, devices, and filesystems.*

Process Management *The two important components in the kernel to perform process management are the* loader*, and the* scheduler*. The loader creates a process out of a program by transferring its contents to memory, and setting up the appropriate execution environment. The* scheduler *schedules the execution of multiple processes including that of the kernel itself.*

Device Drivers *These dedicated programs help the kernel and user processes communicate with devices.*

System Utilities *These are generic services provided by the operating system such as the print queue manager and file manager. They can be used by all the processes in the system.*

Virtual 'View' of Memory

Since multiple processes are live at the same point of time. It is necessary to partition the memory between processes. If this is not done, then it is possible that processes might end up modifying each others' values. At the same time, we do not want the programmer or the compiler to be aware of the existence of multiple processes. This introduces unwanted complexity. Secondly, if a given program is compiled with a certain memory map, it might not run on another machine that has a process with an overlapping memory map. Even worse, it will not be possible to run two copies of the same program. Hence, it is essential that each program sees a virtual view of memory, in which it assumes that it owns the entire memory system.

As we can observe, there are two conflicting requirements. The memory system, and the operating system want different processes to access different memory addresses, whereas, the programmer and the compiler do not want to be aware of this requirement. Additionally, the programmer wishes to layout her memory map according to her wish. It turns out that there is a method to make both the programmer and the operating system happy.

We need to define a *virtual* and a *physical* view of of memory. In the physical view of memory, different processes operate in non-overlapping regions of the memory space. However, in the virtual view, every process accesses any address that it wishes to access, and the virtual views of different processes can overlap. The solution is obtained through a method called *paging* that we shall explain in Section 10.4.3. However, before proceeding to the solution, let us discuss the virtual view of memory that a process typically sees. The virtual view of memory, is also referred to as *virtual memory*. It is defined as a hypothetical memory system, in which a process assumes that it owns the entire memory space, and there is no interference from any other process.

Definition 111
The virtual memory system is defined as a hypothetical memory system, in which a process assumes that it owns the entire memory space, and there is no interference from any other process. The size of the memory is as large as the total addressable memory of the system. For example, in a 32-bit system, the size of virtual memory is 2^{32} *bytes (4 GB). The set of all memory locations in virtual memory is known as the* virtual address space*.*

In the virtual memory space, the operating system lays out the code and data in different regions. This arrangement of code, data, constants, and other information pertaining to a process is known as the *memory map*.

Definition 112
The memory map of a process refers to the way an operating system lays out the code and data in memory.

Memory Map of a Process

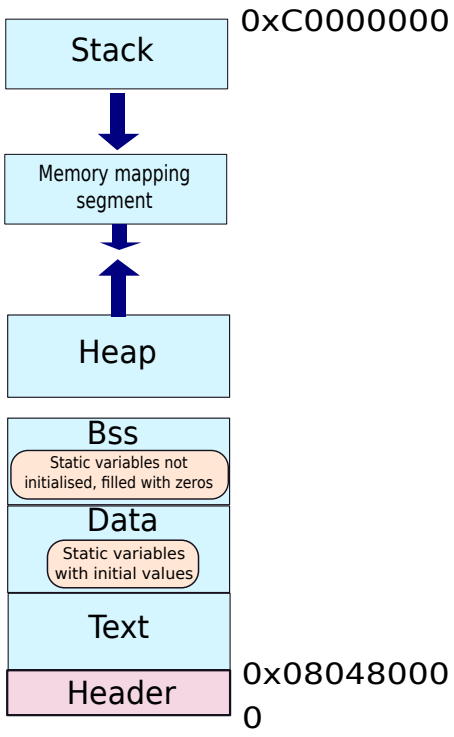


Figure 10.17: Memory map of a process in the Linux operating system (32 bits)

Figure 10.17 shows a simplified view of the memory map of a process in the 32-bit Linux operating system. Let us start from the bottom (lowest address). The first section contains the header. It starts out with details about the process, its format, and the target machine. Subsequently, the header contains the details of each section in the memory map. For example, it contains the details of the text section that contains the code of the program including its size, starting address, and additional attributes. The text section starts after the header. The operating system sets the program counter to the start of the text section while loading a program. All the instructions in a program are typically contained within the text section. The text section is followed by two more sections that are meant to contain static and global variables. Optionally some operating systems, also have an additional area to contain read only data such as constants.

The text section is typically followed by the *data* section. It contains all the static/global variables that have been initialised by the programmer. Let us consider a declaration of the form (in C or C++):

```
static int val = 5;
```

Here the 4 bytes corresponding to the variable – *val* – are saved in the data section. The data section is followed by the *bss* section. The *bss* section saves static and global variables

that have not been explicitly initialised by the programmer. Most operating systems, fill the memory area corresponding to the *bss* section with zeros. This needs to be done in the interest of security. Let us assume that program A runs and writes its values in the *bss* section. Subsequently, program B runs. Before, writing to a variable in the *bss* section, B can always try to read its value. In this case, it will get the value written by program A. However, this is not desirable behavior. Program A might have saved some sensitive data in the bss section such as a password or a credit card number. Program B can thus gain access to this sensitive data without program A's knowledge, and possibly misuse the data. Hence, it is necessary to fill up the *bss* section with zeros such that such kind of security lapses do not happen.

The *bss* section is followed by a memory area known as the *heap*. The *heap* area is used to save dynamically allocated variables in a program. C programs typically allocate new data with the *malloc* call. Java and C++ use the *new* operator. Let us look at some examples.

```
int *intarray = (int *)malloc(10 * sizeof(int));          [C]
int *intarray = new int[10];                              [C++]
int[] intarray = new int[10];                             [Java]
```

Note that in these languages, dynamically allocating arrays is very useful because their sizes are not known at compile time. The other advantage of having data in the heap is that they survive across function calls. The data in the stack remains valid for only the duration of the function call. After that it gets deleted. However, data in the heap stays for the entire life of the program. It can be used by all the functions in the program, and pointers to different data structures in the heap can be shared across functions. Note that the heap grows upward (towards higher addresses). Secondly, managing the memory in a heap is a fairly difficult task. This is because dynamically, regions of the heap are allocated with malloc/new calls and freed with the free/delete calls in high level languages. Once an allocated memory region is freed, a hole gets created in the memory map. It is possible to allocate some other data structure in the hole if its size is less than the size of the hole. In this case, another smaller hole gets created in the memory map. Over time as more and more data structures are allocated and de-allocated, the number of holes tend to increase. This is known as *fragmentation*. Hence, it is necessary to have an efficient memory manager that can reduce the number of holes in the heap. A view of the heap with holes, and allocated memory is shown in Figure 10.18.

The next segment is reserved for storing data corresponding to memory mapped files, and dynamically linked libraries. Most of the time, operating systems transfer the contents of a file (such as a music, text, or video file) to a memory region, and treat the contents of the file as a regular array. This memory region is referred to as a *memory mapped file*. Secondly, programs might occasionally read the contents of other programs (referred to as libraries) dynamically, and transfer the contents of their text sections to their memory map. Such libraries are known as *dynamically linked libraries*, or *dll*s. The contents of such memory mapped structures are stored in a dedicated section in the process's memory map.

The next section is the *stack*, which starts from the top of the memory map and grows downwards (towards smaller addresses) as discussed in Section 3.3.10. The stack continuously grows and shrinks depending on the behavior of the program. Note that Figure 10.17 is not drawn to scale. If we consider a 32-bit memory system, then the total amount of virtual memory is 4 GB. However, the total amount of memory that a program might use is typically limited to

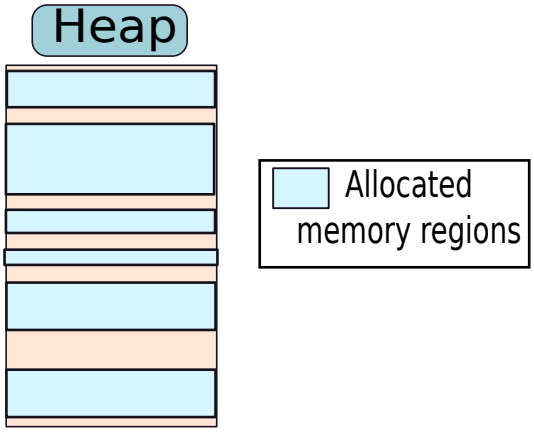


Figure 10.18: The memory map of a heap

hundreds of megabytes. Hence, there is a massive empty region in the map between the start of the *heap* and *stack* sections.

Note that the operating system needs to run very frequently. It needs to service device requests, and perform process management. As we shall see in Section 10.4.3 changing the virtual view of memory from process to process is slightly expensive. Hence, most operating systems partition the virtual memory between a user process and the kernel. For example, Linux gives the lower 3GB to a user process, and keeps the upper 1 GB for the kernel. Similarly, Windows keeps the upper 2GB for the kernel, and the lower 2 GB for user processes. Hence, it is not necessary to change the view of memory as the processor transitions from the user process to the kernel. Secondly, this small modification does not greatly impair the performance of a program because 2GB or 3GB is much more than the typical memory footprint of a program. Moreover, this trick does not also conflict with our notion of virtual memory. A program just needs to assume that it has a reduced memory space (reduced from 4GB to 3GB in the case of Linux). Refer to Figure 10.19.

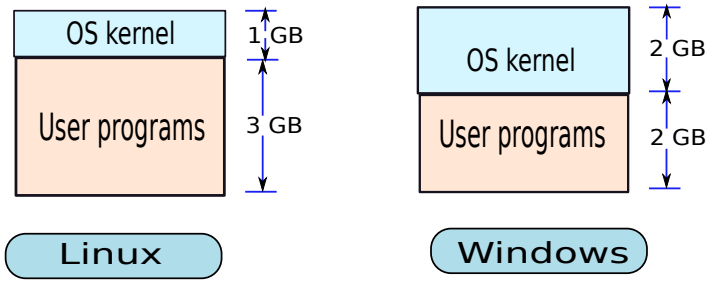


Figure 10.19: The memory map – user and kernel

10.4.2 The "Overlap" and "Size" Problems

Let us summarise all our discussion up till now. We basically want to solve two problems.

Overlap Problem Programmers and compilers write a program assuming that they own the entire memory space and they can write to any location at will. Unfortunately, the same assumption is made by all processes that are simultaneously active. Unless steps are taken, they may end up inadvertently writing to each other's memory space and corrupting each other's data. In fact, given that they use the same memory map, the chances of this happening in a naive system are extremely high. The hardware somehow needs to ensure that different processes are isolated from each other. This is the *overlap problem*.

Size Problem Occasionally we need to run processes that require more memory than the available physical memory. It is desirable if some space in other storage media such as the hard disk can be repurposed for storing the memory footprint of a process. This is known as the *size problem*.

Any implementation of virtual memory needs to effectively solve the size and overlap problems.

10.4.3 Implementation of Virtual Memory with Paging

To balance the requirements of the processor, operating system, compiler, and programmer we need to design a translation system that can translate the address generated by a process into an address that the memory system can use. By using a translator, we can satisfy the requirements of the programmer/compiler, who need virtual memory, and the processor/memory system, who need physical memory. A translation system is similar to what a translator in real life would do. For example, if we have a Russian delegation visiting Dubai, then we need a translator who can translate Russian to Arabic. Both the sides can then speak their own language, and thus be happy. A conceptual diagram of the translation system is shown in Figure 10.20.

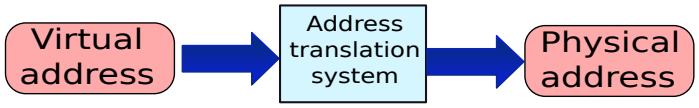


Figure 10.20: Address translation system

Let us now try to design this address translation system. Let us first succinctly list the requirements that a program and compiler place on the nature of virtual memory.

1. Any address in the range of valid addresses should be accessible. For example, in a Linux based machine, a process's virtual memory size is limited to 3 GB. Hence, it should be possible to access any address in this range.

2. The virtual memory should be perceived as one contiguous memory space where the entire space is available to the program.

3. Unless explicitly desired by the program, there should be no interference from any other program.

Here are the requirements from the side of the memory system.

1. Different programs should access non-overlapping sets of addresses.
2. A program cannot be allotted a large continuous chunk of memory addresses. This will cause a high degree of wastage in space due to fragmentation.
3. If the total amount of physical memory is less than the size of the virtual memory, then there should be additional storage space available to support programs that require more space than the total amount of physical memory.

Let us now try to satisfy these requirements by designing a translation system that takes an address as specified in the program, and translates it to a real address that can be presented to the memory system. The address specified in the program is known as the *virtual address*, and the address sent to the memory system is known as the *physical* address.

Definition 113

Virtual Address *An address specified by the program in the virtual address space.*

Physical Address *An address presented to the memory system after address translation.*

We can trivially achieve a translation system by uniquely mapping every virtual address to a physical address at the level of every byte or memory word (4 bytes). In this case, the program perceives one contiguous memory space. Secondly, we need to only map those virtual addresses that are actually used by the program. If a program actually requires 3 MB of space, then we end up using only 3 MB of physical memory. Whenever, the process requires a new set of bytes that have not been already mapped, a smart memory management unit can allocate new space in physical memory. Lastly, note that it is necessary for every memory access to pass through this translation system.

Even though our basic translation system satisfies all our requirements, it is not efficient. We need to maintain a large table that maps every byte in the virtual address space to a byte in the physical address space. This mapping table between the virtual and physical addresses will be very large and slow. It is also not a very power efficient scheme. Secondly, our scheme does not take advantage of spatial and temporal locality. Hence, let us try to make our basic system more efficient.

Pages and Frames

Definition 114

Page *It is a block of memory in the virtual address space.*

Frame *It is a block of memory in the physical address space. A page and frame have the same size.*

Page Table *It is a mapping table that maps the address of each page to an address of a frame. Each process has its own page table.*

Instead of translating addresses at the granularity of bytes, let us translate addresses at the granularity of larger blocks. This will reduce the amount of state that we need to maintain, and also take advantage of spatial locality. Let us define a block of memory in the virtual address space and call it a *page*. Similarly, let us define a block of the same size in the physical address space and call it a *frame*. The size of a page or a frame is typically 4 KB. Secondly, note that the virtual address space is unique to each process; whereas, the physical address space is the same for all processes. For each process, we need to maintain a mapping table that maps each page to a frame. This is known as the *page table*. A page table can either be implemented in hardware or in software. A hardware implementation of the page table has dedicated structures to store the mapping between virtual and physical addresses. The lookup logic is also in hardware. In the case of a software implementation, the mappings are stored in a dedicated region of the physical address space. In most processors that use software page tables, the lookup logic is also in hardware. They typically do not use custom routines in software to lookup page tables because this approach is slow and complicated. Since the lookup logic of page tables is primarily in hardware, the design of page tables needs to be relatively simple. The page tables that we describe in the next few sections are oblivious to how they are implemented (software or hardware).

Let us consider a 32-bit memory address. We can now split it into two parts. If we consider a 4 KB page, then the lower 12 bits specify the address of a byte in a page (reason: $2^{12} = 4096 = 4KB$). This is known as the *offset*. The upper 20 bits specify the *page number* (see Figure 10.21). Likewise, we can split a physical address into two parts – frame number and offset. The process of translation as shown in Figure 10.21, first replaces the 20 bit page number with an equivalent 20 bit frame number. Then it appends the 12 bit offset to the physical frame number.

A Single Level Page Table

Figure 10.22 shows a basic page table that contains 2^{20} ($\approx 1,000,000$) rows. Each row is indexed by the page number, and it contains the corresponding 20 bit (2.5 byte) frame number. The total size of the table is thus 2.5 MB. If we have 200 processes in the system at any point of time, then we need to waste 500 MB of precious memory for just saving page tables! If our total main memory is 2 GB, then we are spending 25% of it in saving page tables, which appears to be a big waste of space. Secondly, it is possible that in some systems, we might not even have

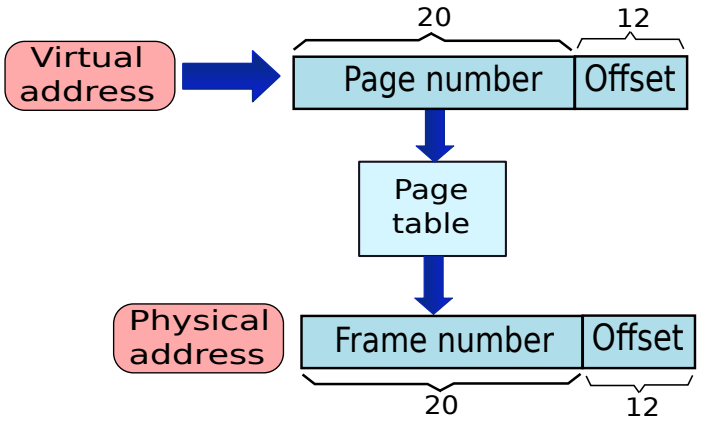


Figure 10.21: Translation of a virtual to a physical address

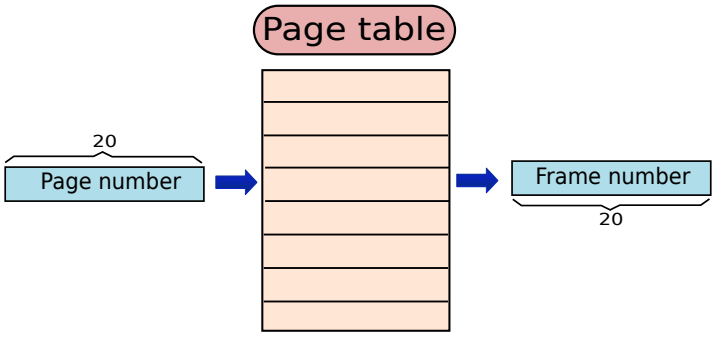


Figure 10.22: A single level page table

500 MB of main memory available. In this case, we cannot support 200 live processes at the same time. We need to look for better solutions.

Let us now look for insights that might help us reduce the amount of storage. We start out by noticing that large parts of the virtual address space of a process are actually empty. In a 32-bit system, the size of the virtual address space is 4 GB. However, large programs do not use more than 100 MB. There is a massive empty region between the stack and the heap sections in the memory map, and thus it is not necessary to allocate space for mapping this region. Ideally, the number of entries in the page table should be equal to the number of pages actually used by a process rather than the theoretically maximum number of pages a process can use. If a process uses only 400 KB of memory space, then ideally its page table should just contain 100 entries. Let us design a two level page table to realise this goal.

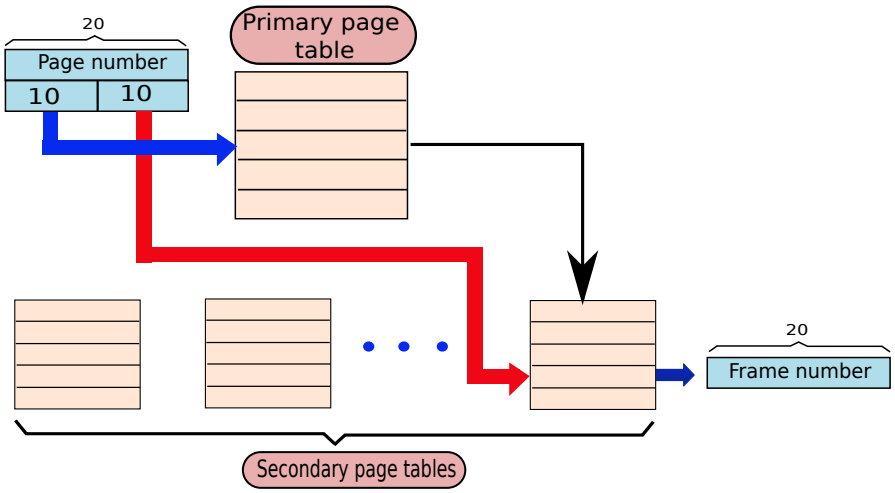


Figure 10.23: A two level page table

Two Level Page Table

Let us further split a page number into two equal parts. Let us split the 20 bits into two parts containing 10 bits each as shown in Figure 10.23. Let us use the upper 10 bits to access a top level page table known as the *primary page table*. Each entry in the top level page table points to a secondary page table. Subsequently, each secondary page table is indexed by the lower 10 bits of the page number. An entry in the secondary page table contains the frame number. If no addresses map to a given entry in the primary page table, then it does not point to a secondary page table, and thus there is no need to allocate space for it. In a typical program, most of the entries in the primary page table are expected to be empty. Let us now calculate the size of this structure.

The primary page table contains 1024 entries, where each entry is 10 bits long. The total size is 1.25 KB (10 bits = 1.25 bytes). Let the number of secondary page tables be N. Each secondary page tables contains 1024 entries, where each entry is 20 bits long. Therefore, the size of each secondary page table is 2.5 KB, and the total storage requirement is $(1.25+2.5\times N)$ KB. Because of spatial locality in a program, N is not expected to be a large number. Let us consider a program that has a memory footprint of 10 MB. It contains roughly 2500 pages. Each secondary page table can map at the most 1024 pages (4 MB of data). It is highly likely that this program might map to only 3 secondary page tables. Two page tables will contain the mappings for the *text*, *data*, and *heap* sections, and one page table will contain the mappings for the *stack* section. In this case, the total storage requirement for the page tables will be equal to 8.75 KB, which is very reasonable. Even, if we require double the number of secondary page tables because of lower spatial locality in the memory map, then also the total storage requirement is equal to 16.25 KB. This is an order of magnitude better than a single level page table that required 2.5 MB of storage per process. Hence, two level page tables are used in most commercial systems.

Inverted Page Table

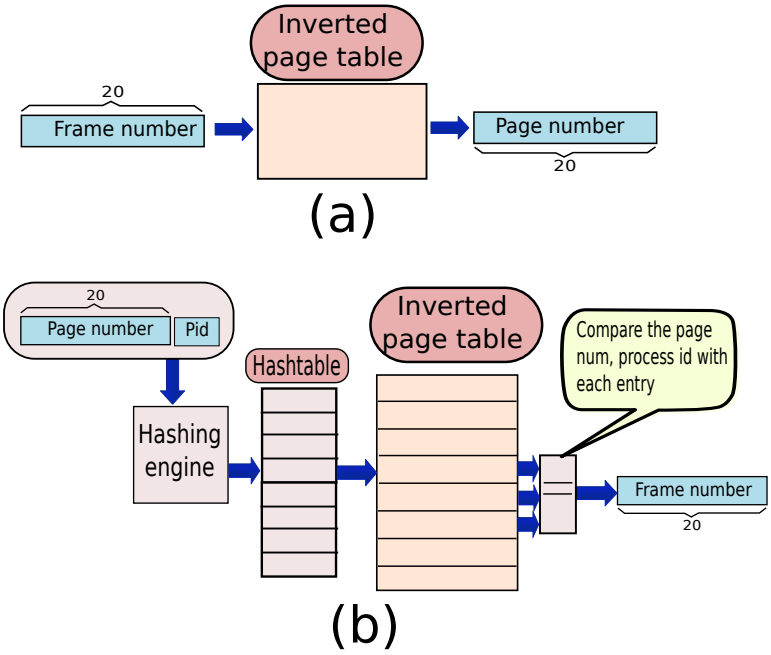


Figure 10.24: Inverted page table

Some processors such as the Intel Itanium, and PowerPC 603, use a different design for a page table. Instead of addressing the page table using the page number, they address it using the frame number. In this case, there is one page table for the entire system. Since one frame is typically uniquely mapped to a page in a process, each entry in this *inverted page table* contains the process id, and page number. Figure 10.24(a) shows the structure of an inverted page table. The main advantage of an inverted page table is that we do not need to keep a separate page table for each process. We can save space if there are a lot of processes, and the size of physical memory is small.

The main difficulty in inverted page tables is in performing a lookup for a virtual address. Scanning all the entries is a very slow process, and is thus not practical. Hence, we need to have a hashing function that maps the (process id, page number) pair to an index in a hash table. This index in the hash table needs to point to an entry in the inverted page table. Since multiple virtual addresses can point to the same entry in the hash table, it is necessary to verify that the (process id, page number) matches that stored in the entry in the inverted page table. Readers can refer to [Cormen et al., 2009] for a detailed explanation of the theory and operation of hash tables.

We show one scheme for using an inverted page table in Figure 10.24(b). After computing a hash of the page number, and process id pair, we access a hashtable indexed by the contents of the hash. The contents of the hashtable entry point to a frame, f, that might possibly map to the given page. However, we need to verify, since it is possible that the hash function maps

multiple pages to the same frame. Subsequently, we access the inverted page table, and access the entry, f. An entry of the inverted page table, contains the page number, process id pair that is mapped to the given entry (or given frame). If we find that the contents do not match, then we keep searching for the page number, process id pair in the subsequent K entries. This method is called linear probing (see [Cormen et al., 2009]), where we keep searching in the target data structure till we get a match. If we do not get a match within K entries, then we may conclude that the page is not mapped. We need to then create a mapping, by evicting an entry (similar to caches), and writing it to a dedicated region in main memory that buffers all the entries that are evicted from the inverted page table. We need to always guarantee that the entry pointed to by the hash table, and the actual entry that contains the mapping, do not differ by more than K entries. If we do not find any free slots, then we need to evict an entry.

An astute reader might argue that we can directly use the output of the hashing engine to access the inverted page table. Typically, we add accessing a hashtable as an intermediate step, because it allows us to have better control over the set of frames that are actually used. Using this process, it is possible to disallow mappings for certain frames. These frames can be used for other purposes. Lastly, we need to note that the overhead of maintaining, and updating hash tables outweighs the gains in having a system wide page table. Hence, an inverted page table is typically not used in commercial systems.

Translation Lookaside Buffer (TLB)

For every single memory access it is necessary to lookup the page table for translating the virtual address. The page table itself is stored in physical memory. Hence, we need to do a separate memory access to read the corresponding entry of the page table. This approach doubles the number of memory accesses, and is thus very inefficient. However, we can minimise the number of extra memory accesses by maintaining a small cache of mappings in the processor. We typically use a structure known as the Translation Lookaside Buffer (TLB) that is a small fully associative cache. A TLB contains 32-64 entries. Each entry is indexed by the page number, and contains the corresponding frame number.

Once a memory address is calculated in the EX stage of the pipeline. It is sent to the TLB. The TLB is a very fast structure, and typically its access time is a fraction of a cycle. If there is a TLB hit, then the physical address is ready by the time we reach the memory access (MA) stage. The MA stage of the pipeline can then issue the read/write request to the memory system using the physical address obtained from the TLB. However, if there is a TLB miss, then the pipeline needs to stall, and the page table needs to be accessed. This is a slow process and takes tens of cycles. Fortunately, the hit rate of a TLB is very high ($\approx 99\%$) in most programs because of two reasons. First, programs have a high degree of temporal locality. Second, a 64 entry TLB covers 256 KB of the virtual address space (assuming a 4 KB page). The working set of most programs fits within this limit for small windows of time.

10.4.4 Swap Space

We have solved the first problem, i.e., ensuring that processes do not overwrite each other's data. Now, we need to solve the second problem, which is to ensure that our system can run even when the memory footprint of a program is more than the amount of physical memory. For example, we might need to run a program with a memory footprint of 3 GB on a machine

with only 512 MB of main memory. Even on regular desktop machines it is possible that the combined memory footprint of all the processes is more than the size of main memory.

To support this requirement, we first need to find a location to save all the data that does not fit in main memory. Most processors typically have peripheral devices connected to the processor such as the hard disk, or USB flash drives that have a large amount of storage capacity. We shall study about storage devices in detail in Chapter 12. In this section, we only need to appreciate the following aspects of such connected storage devices.

1. Connected storage devices are very slow as compared to main memory. The access time to main memory is about 100-300 ns; whereas, the access time to a hard disk is of the order of milliseconds.

2. Storage devices typically have several orders of magnitude more storage than main memory. A hard disk contains about 500 GB of storage in a system with 4 GB of main memory.

3. They are conceptually treated as a large array of bytes similar to the way we treat the memory system. However, an address in the memory system is unrelated to the address in a hard disk. The storage device is **not a part of the memory system**.

4. It is not necessary to have a storage device physically close to the processor. It can be accessible over the network, and be in another part of the world.

A storage device can define an area known as the *swap space* that has space to contain all the frames that cannot be saved in main memory. Furthermore, this storage region need not be a superset of the main memory. If it is an extension of main memory, then we can define a larger physical memory. For example, if we have 2 GB of main memory, and 3 GB of swap space, then the total amount of physical memory can be 5 GB. In this case, if we need to displace a frame from main memory, then we need to allocate a location for it in swap space. Alternatively, the swap space can be inclusive. In the above example, we will effectively have 3 GB of physical memory, and the main memory acts like a cache for the swap space. In either case, the role of the swap space is to increase the amount of available physical memory.

Now, the obvious question that arises is, "How does the memory system know if a frame is present in main memory or the swap space? " We can augment each entry in the page table with an extra bit. If this bit is 1, then the frame is in main memory, else it is in the swap space. Note that this system can be made more complicated also. Instead of one swap space, we can have multiple swap spaces, and use multiple bits in a page table entry to indicate the corresponding swap space.

10.4.5 Memory Management Unit (MMU)

Up till now we have not discussed how page tables are actually managed and stored. Let us consider the typical life cycle of a process. When a process begins, the kernel allocates a primary page table in main memory, and clears off the TLB. It can then insert the mappings for the *text*, and *data*, sections. Secondly, the kernel can optionally allocate some space and insert some mappings for the *heap*, and *stack* sections. As long as there is a TLB hit, there is no problem. Once, there is a TLB miss, it is necessary to access the page tables, and secondly, the pipeline

needs to stall. The job of accessing the page tables is typically handled by a dedicated unit known as the MMU (memory management unit). It can either be a hardware structure, or a software structure. If it is a hardware structure, then we have dedicated logic in the processor. Otherwise, it is necessary to invoke the MMU process by suspending the current process.

In either case, the operation of the MMU is the same. It needs to first locate the starting address of the primary page table. Note that this address cannot be a virtual address. Otherwise, we will need a page table for a page table. It is typically a physical address that does need additional translation. This starting address is either kept in a dedicated processor register (CR3 in x86), or in a designated location in physical memory. The MMU then needs to access the appropriate entry in the primary page table, and get the address of the secondary page table. The address of the secondary page table is another physical address. If a secondary page table exists, then the MMU accesses the relevant entry in the secondary page table, and gets the frame number. Subsequently, it evicts an entry from the TLB, and adds the new mapping. It can follow a LRU replacement scheme as described in Section 10.2.5. Note that it is necessary to have all the page tables in the main memory. They cannot be in the swap space.

Page Fault

There are several things that can go wrong in this process. If a page is being accessed for the first time, it is possible that it might not have a secondary page table, or its corresponding entry in the secondary page table might be empty. In this case, it is necessary to first find a free frame in main memory, create a secondary page table if required, and then insert the mapping in the secondary page table. To find a free frame in memory the MMU must maintain information about each frame. This information can be kept in the form of a bit vector, where each bit corresponds to a frame in main memory. If it is free, then the bit is 0, else if it is mapped, the bit is 1. If a free frame is available, then it can be used to map the new page. Otherwise, we need to forcibly free a frame by writing its data to the swap space. The method of finding a frame to evict from main memory is known as the *page replacement policy*. Subsequently, we need to change the page table entry of the page that was previously mapped to this frame. It needs to now say that the page is available in swap space. Once a frame has been freed, it can be mapped to another page.

Alternatively, it is also possible that the entry in the page table indicates that the frame is there in swap space. In this case, it is necessary to bring the frame into main memory. We first need to find a free frame, or if necessary evict a frame from main memory. Then, we need to create an appropriate page table mapping.

Definition 115
Whenever a page is not found in main memory, the event is known as a page fault*.*

Whenever a page is not found in main memory, we term the event as a *page fault*. It is subsequently necessary to create appropriate mappings in the page table and fetch the data from the swap space. Fetching an entire page from the swap space is a rather slow operation, and takes millions of cycles. Hence, it is very important for the MMU to manage pages efficiently.

In specific, the page fault rate is very sensitive to the page replacement policy. Similar to cache block replacement policies, we can have different kinds of page replacement policies such as FIFO (first in first out), and LRU (least recently used). For more information on page replacement policies, the reader is referred to a textbook on operating systems [Silberschatz et al., 2008, Tanenbaum, 2007].

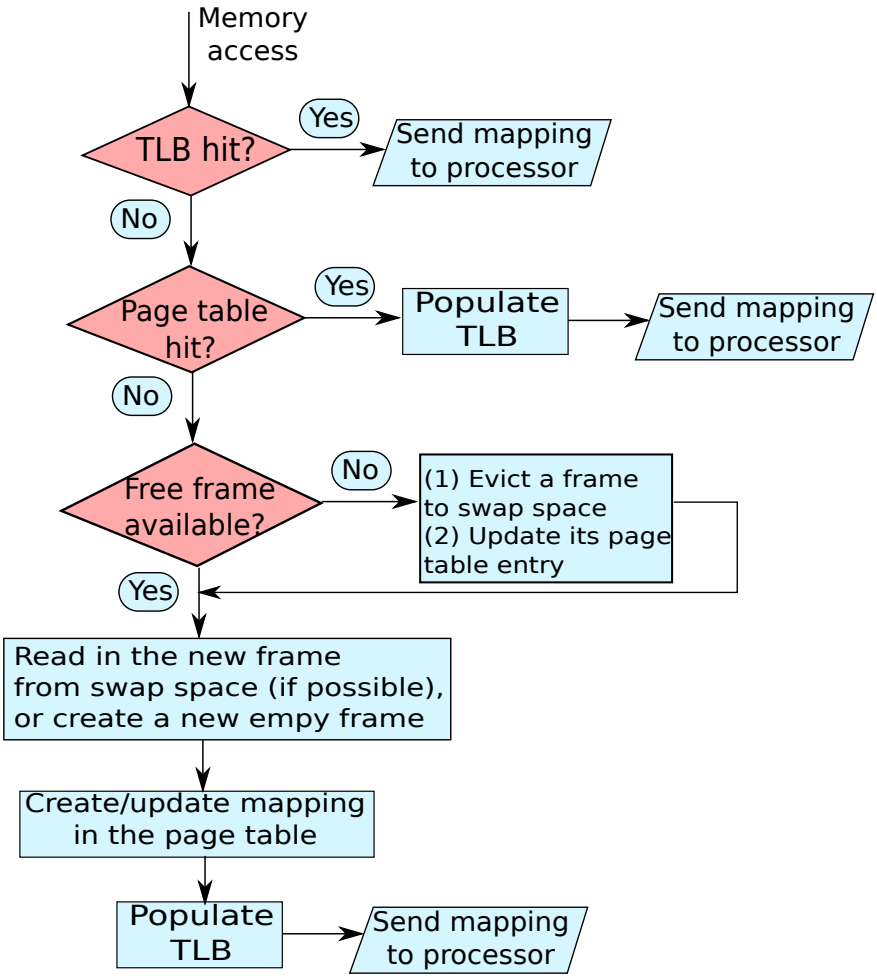


Figure 10.25: The process of address translation

Figure 10.25 summarises the major steps in the process of address translation.

10.4.6 Advanced Features of the Paging System

It turns out that we can do several interesting things with the page table mechanism. Let us look at a few examples.

Shared Memory

Let us assume that two processes want to share some memory between each other such that they can exchange data between them. Then each process needs to let the kernel know about this. The kernel can then map two pages in both the virtual address spaces to the same frame. Now, each process can write to a page in its own virtual address space, and magically, the data will get reflected in the virtual address space of the other process. It is sometimes necessary for several processes to communicate among each other, and the shared memory mechanism is one of the fastest methods.

Protection

Computer viruses typically change the code of a running process such that they can execute their own code. This is typically achieved by a giving a specific sequence of erroneous inputs to the program. If appropriate checks are not in place, then the values of specific variables within the program get overwritten. Some variables can get changed to pointers to the *text* section, and it is possible to exploit this mechanism to change instructions in the *text* section. It is possible to solve this problem by marking all the pages in *text* section as read-only. It will thus not be possible to modify their contents in run time.

Segmentation

We have been assuming that a programmer is free to layout the memory map according to her wish. She might for example decide to start the stack at a very high address such as $0xFFFFFFF8$. However, this code might not run on a machine that uses 16-bit addresses even if the memory footprint of the program is very small. Secondly, it is possible that a certain system might have reserved some parts of the virtual memory and made them unavailable to the process. For example, operating systems typically reserve the upper 1 or 2 GB for the kernel. To solve these problems, we need to create another virtual layer on top of virtual memory.

In a segmented memory (used in x86 systems), there are specific segment registers for the *text*, *data*, and *stack* sections. Each virtual address is specified as an offset to the specific segment register. By default instructions use the code segment register, and data uses the data segment register. The memory access (MA) stage of the pipeline adds the offset to the value stored in the segment register to generate the virtual address. Subsequently, the MMU uses this virtual address to generate the physical address.

10.5 Summary and Further Reading

10.5.1 Summary

Summary 10

1. *A program perceives the memory system to be one large array of bytes. In practice, we need to design a memory system that preserves this abstraction, and is also fast and power efficient.*

2. *A physical memory system needs to be built out of SRAM and DRAM cells. An SRAM array is faster than a DRAM array. However, it takes much more area and consumes much more power. Building a memory with just DRAM cells will be too slow, and building a memory with just SRAM cells will be consume too much power.*

3. *We can use the properties of temporal and spatial locality to design more efficient memory systems. Temporal locality refers to the fact that there is a high likelihood of the same data item being accessed again in the near future. Spatial locality means that there is a high likelihood of adjacent memory locations being accessed in the near future.*

4. *To utilise temporal locality, we build a hierarchical memory system of caches. A cache is a memory structure that contains a subset of all the memory locations.*

   (a) *The cache at the highest level is known as the L1 cache. It is small and fast.*

   (b) *The L2 cache is at the next level. It is larger and slower.*

   (c) *Some recent processors also have a third level of cache known as the L3 cache.*

   (d) *The last level in the memory system is known as the* main memory*. It is a large DRAM array of cells, and contains an entry for all the memory locations in the system.*

   (e) *Caches are typically inclusive. This means that a cache at a level i contains a subset of memory locations present at level $(i+1)$.*

5. *To utilise spatial locality we group adjacent memory locations at the granularity of 32-128 byte blocks.*

6. *A cache contains a tag array and a data array. The tag array contains some of the bits of the address of the block, and the data array contains the contents of the block.*

7. *The basic operations needed to implement a cache are –* lookup*,* data read*,* data write*,* insert*,* replace*, and* evict*.*

   (a) *There are three ways to store data in a cache – direct mapped, set associative, and fully associative.*

   (b) *It is necessary to evict a block in a set if all the ways are non-empty.*

   (c) *There are two major write policies – write-through (every write is immediately sent to the lower level), and write-back (writes are sent to the lower level, only upon an eviction)*

   (d) *Some of the prominent replacement policies are – Random, FIFO, and LRU.*

8. *The average memory access time is given by:*

$$\begin{aligned} AMAT &= L1_{hit\,time} + L1_{miss\,rate} \times L1_{miss\,penalty} \\ &= L1_{hit\,time} + L1_{miss\,rate} \times (L2_{hit\,time} + L2_{miss\,rate} \times L2_{miss\,penalty}) \end{aligned}$$

9. *There are three types of cache misses – compulsory, capacity, and conflict.*

10. *Some of the methods and structures to optimise the memory system are: hardware prefetching, increased associativity/block size, victim cache, compiler techniques, write buffers, early restart and critical word first.*

11. *We need virtual memory to ensure that:*

    (a) *Multiple programs do not overwrite each other's data unintentionally, or maliciously.*

    (b) *The memory footprint of a program can be larger than the amount of available main memory.*

12. *To implement virtual memory, we divide a memory address into two parts – virtual page number, and an offset within a page. The virtual page number gets mapped to a physical frame number. The mapping is stored in a structure called a page table.*

13. *If a page is not found in main memory, then the event is known as a* page fault*. Servicing a page fault takes millions of cycles. Hence, it is necessary to avoid page faults by using sophisticated page replacement algorithms.*

14. *Some of the advanced features of the virtual memory system include shared memory, protection, and segmented addressing.*

10.5.2 Further Reading

The reader can refer to advanced text books on computer architecture by Henessey and Patterson [Hennessy and Patterson, 2012], Kai Hwang [Hwang, 2003], and Jean Loup Baer [Baer, 2010] for a discussion on advanced memory systems. Specifically, the books discuss advanced techniques for prefetching, miss rate reduction, miss penalty reduction, and compiler directed approaches. The reader can also refer to the book on memory systems by Bruce Jacob [Jacob, 2009]. This book gives a comprehensive survey of most of the major techniques employed in designing state of the art memory systems till 2009. The book by Balasubramaniam, Jouppi, and Muralimanohar on cache hierarchies also discusses some of the more advanced topics on the management of caches [Balasubramonian et al., 2011]. Managing the MMU is mostly studied in courses on operating systems [Tanenbaum, 2007, Silberschatz et al., 2008]. Research in DRAM memories [Mitra, 1999], and systems using advanced memory technologies is a hot topic of current research. A lot of research work is now focusing on phase change memories that do not require costly refresh cycles like DRAM. Readers can refer to the book by Qureshi, Gurumurthi, and Rajendran [Qureshi et al., 2011] for a thorough explanation of memory systems using phase

change memories.

Exercises

Overview

Ex. 1 — Define temporal locality, and spatial locality.

Ex. 2 — Experimentally verify that the log-normal distribution is a heavy tailed distribution. What is the implication of a heavy tailed distribution in the context of the stack distance and temporal locality?

Ex. 3 — Define the term, *address distance*. Why do we find the nearest match in the last K accesses?

Ex. 4 — How do we take advantage of temporal locality in the memory system?

Ex. 5 — How do we take advantage of spatial locality in the memory system?

Caches and the Memory System

Ex. 6 — Consider a fully associative cache following the LRU replacement scheme and consisting of only 8 words. Consider the following sequence of memory accesses (the numbers denote the word address):
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 22, 30, 21, 23, 31

Assume that we begin when the cache is empty. What are the contents of the cache after the end of the sequence of memory accesses.

Ex. 7 — Answer Exercise 6 assuming a FIFO replacement scheme.

Ex. 8 — Consider a two-level cache using a write back policy. The L1 cache can store 2 words, and the L2 cache can store 4 words. Assume the caches to be fully associative (block size = 1 word); they follow the LRU replacement scheme. Consider the following sequence of memory accesses. The format of a write access is *write* <*address*> <*value*>, and the format for a read access is *read* <*address*> .

```
write 20 200
write 21 300
write 22 400
write 23 500
write 20 201
write 21 301
read 22
read 23
```

```
write 22 401
write 23 501
```

What are the contents of the caches at the end of the sequence of memory accesses? What are the contents of the caches, if we assume a write through policy ?

Ex. 9 — What is the total size (in bytes) of a direct mapped cache with the following configuration in a 32 bit system? It has a 10 bit index, and a block size of 64 bytes. Each block has 1 valid bit and 1 dirty bit.

Ex. 10 — Which sorting algorithm will have a better cache performance – bubble sort or selection sort? Explain your answer.

Ex. 11 — You have a cache with the following parameters:

- size : n bytes
- associativity : k
- block size : b bytes

Assuming a 32-bit address space, answer the following:

(a) What is the size of the tag in bits?

(b) What is the size of the set index in bits?

* **Ex. 12** — Consider a direct mapped cache with 16 cache lines, indexed 0 to 15, where each cache line contains 32 integers (block size : 128 bytes).
Consider a two-dimensional, 32×32 array of integers a. This array is laid out in memory such that $a[0,0]$ is next to $a[0,1]$, and so on. Assume the cache is initially empty, and $a[0,0]$ maps to the first word of cache line 0.
Consider the following *column-first* traversal:

```
int sum = 0;
for (int i = 0; i < 32; i++) {
    for( int j=0; j < 32; j++) {
          sum += a[i,j];
    }
}
```

and the following *row-first* traversal:

```
int sum = 0;
for (int i = 0; i < 32; i++) {
    for( int j=0; j < 32; j++) {
         sum += a[j,i];
    }
}
```

Compare the number of cache misses produced by the two traversals, assuming the oldest cache line is evicted first. Assume that i, j, and sum are stored in registers, and that no part of array a is saved in registers. It is always stored in the cache.

Ex. 13 — A processor has a baseline IPC of 1.5, an L1 miss rate of 5%, and an L2 miss rate of 50%. The hit time of the L1 cache is 1 cycle (part of the baseline IPC computation), the L2 hit time is 10 cycles, and the L2 miss penalty is 100 cycles. Compute the final IPC. Assume that all the miss rates are local miss rates.

Ex. 14 — Consider the designs shown below

Design	Base CPI	L1 local miss rate (%)	L2 local miss rate (%)	L1 hit time (cycles)	L2 hit time (cycles)	L2 miss penalty (cycles)
$\mathcal{D}_1$	1	5	20	1	10	200
$\mathcal{D}_2$	1.5	10	25	1	20	150
$\mathcal{D}_3$	2	15	20	1	5	300

The base CPI assumes that all the instructions hit in the L1 cache. Furthermore, assume that a third of the instructions are memory instructions.
Write the formula for the average memory access time. What is the CPI of $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$?

* **Ex. 15** — Assume a cache that has n levels. For each level, the hit time is x cycles, and the local miss rate is y per cycle.

(a) What is the recursive formula for the average memory access time?

(b) What is the average memory access time as n tends to ∞?

** **Ex. 16** — Assume that you are given a machine with an unknown configuration. You need to find out a host of cache parameters by measuring the time it takes to execute different programs. These programs will be tailor made in such a way that they will reveal something about the underlying system. For answering the set of questions, you need to broadly describe the approach. Assume that the cache follows the LRU scheme for replacement.

(a) How will you estimate the size of the L1 cache?

(b) How will you estimate the L1 block size?

(c) How will you estimate the L1 cache associativity?

Virtual Memory

Ex. 17 — In a 32-bit machine with a 4 KB page size, how many entries are there in a single level page table? What is the size of each entry in the page table in bits?

Ex. 18 — Consider a 32-bit machine with a 4 KB page size, and a two level page table. If we address the primary page table with 12 bits of the page address, then how many entries are there in each secondary page table?

Ex. 19 — In a two level page table, should we index the primary page table with the most significant bits of the page address, or the least significant bits? Explain your answer.

Ex. 20 — We have a producer-consumer interaction between processes A and B. A writes data that B reads in a shared space. However, B should never be allowed to write anything into that shared space. How can we implement this using paging? How do we ensure that B will never be able to write into the shared space?

Ex. 21 — Assume a process A, forks a process B. Forking a process means that B inherits a copy of A's entire address space. However, after the fork call, the address spaces are separate. How can we implement this using our paging mechanism?

Ex. 22 — How is creating a new thread different from a fork() operation in terms of memory addressing?

Ex. 23 — Most of the time, the new process generated by a $fork$ call does not attempt to change or modify the data inherited from the parent process. So is it really necessary to copy all the frames of the parent process to the child process? Can you propose an optimisation?

Ex. 24 — Explain the design of an inverted page table.

* **Ex. 25** — Calculate the expected value of the final CPI:

- Baseline CPI: 1
- Percentage of memory accesses: 30%
- TLB lookup time: 1 cycle (part of the baseline CPI)
- TLB miss rate: 20%
- Page table lookup time: 20 cycles (do not assume any page faults). Assume we can instantaneously insert entries into the TLB.
- L1 cache hit time: 1 cycle (Part of the baseline CPI)
- L1 local miss rate: 10%
- L2 cache hit time: 20 cycles
- L2 local miss rate: 50%
- L2 miss penalty: 100 cycles

** **Ex. 26** — Most of the time, programmers use libraries of functions in their programs. These libraries contain functions for standard mathematical operations, for supporting I/O operations, and for interacting with the operating system. The machine instructions of these functions are a part of the final executable. Occasionally, programmers prefer to use dynamically linked libraries (DLLs). DLLs contain the machine code of specific functions. However, they are invoked at run time, and their machine code is not a part of the program executable. Propose a method to implement a method to load and unload DLLs with the help of virtual memory.

Design Problems

Ex. 27 — You need to learn to use the CACTI tool (`http://www.hpl.hp.com/research/cacti/`) to estimate the area, latency, and power of different cache designs. Assume a 4-way associative 512 KB cache with 64 byte blocks. The baseline design has 1 read port, and 1 write port. You need to assume the baseline design, vary one parameter as mentioned below, and plot its relationship with the area, latency, or power consumption of a cache.

a)Plot the area versus the number of read ports.

b)Plot the energy per read access versus the number of read ports.

c)Plot the cache latency versus the associativity.

d)Vary the size of the cache from 256 KB to 4 MB (powers of 2), and plot its relationship with area, latency, and power.

Ex. 28 — Write a cache simulator that accepts memory read/write requests and simulates the execution of a hierarchical system of caches.