Week 13-14 Module 21, Part 2 – Computer Organization

Outline 1.2

Hi everyone, today we will cover the second part of the computer organization section of this course. We will focus on the memory hierarchy, which is one of the most important aspects of CPU design. We will discuss the need for caches, which are smaller memory components between the CPU and the main memory, how they are organized and how they facilitate faster access to code and data during execution

Memory Hierarchy 1.3

So far, we have assumed that every memory access that the CPU makes takes one clock cycle. That is, each time an instruction is fetched, or a data is written to or read from memory, it only requires one cycle. However, in reality, each memory access takes several cycles. This is because of two reasons. The first and most important reason is that main memory is generally implemented as DRAM, which stands for dynamic random access memory. In this instance, dynamic means that the data needs to be refreshed from time to time, in order to keep it available in memory. This refreshing delays the access time of data because it may need to be refreshed before it is brought to the CPU. The second reason is that the main memory is so large, usually in the gigabyte range, that it has to be put off the CPU chip, causing it to run at a lower frequency than the CPU itself. Therefore, the main memory is running at a slower speed than the CPU, causing additional delays.

In many cases, access to main memory can take as many as 100 cycles for each access. Therefore, if we consider a simple pipelined CPU, each instruction fetch stage requires 100 cycles. Similarly, each memory stage where the instruction reads from memory or writes to memory also requires 100 cycles. In this small example, we can see how the execution time goes from 7 to at least 700 cycles when we consider the real access delay of memory.

Caches 1.4

The primary approach to overcome the main memory access delay is by adding caches. A cache is another memory component that is put between the main memory and the processor core and its role is to hold instructions and data that the CPU often uses. So what makes a cache different from DRAM? First off, a cache is implemented as SRAM instead of DRAM. The S in SRAM stands for static. This means that there is no need to refresh the main memory, which makes it faster to read from and write to. However, SRAM requires more space to hold one bit of data than DRAM. Usually, SRAM can require up to 8 times more area to store one bit than a DRAM. So, although it consumes less power, SRAM takes more area. As a result, we can't have gigabytes worth of SRAM. Usually, an SRAM cache is in the range of 2KB to 2-MB. Given that a cache can be in the kilobyte size, it is small enough that it can be put on the same chip as the CPU without it consuming all of the power budget. So with the benefits of using static RAM and being able to put the cache on the same chip as the CPU, the access delay of a cache can be between 1 and 12 cycles depending on the size and other factors that we will discuss. Now someone

may have the intuition to simply use SRAM for main memory instead of DRAM. But remember, SRAM can require up to 8 more area than DRAM. Therefore, you cannot have a large 8 GB main memory as SRAM because it will take too much space on your computer motherboard.

Caches 1.5

So this is a generalized view of the memory hierarchy that incorporates the cache. This memory hierarchy is representative of the hierarchy found in a typical laptop. So the idea is the following: starting from the bottom of the triangle, the higher you go, the less is the access delay, but the smaller the memory gets. Therefore, we get a trade-off between speed and size of the memory.

At the lowest level, we have the hard disk, which is nowadays in the range of terabytes and takes millions of cycles to access. To put it in perspective, if we have a standard laptop, it takes 0.5 milliseconds to access the disk, which is pretty slow by processor performance standards. At the next level, we have the main memory, which can up to 8 gigabyte of DRAM nowadays. A typical access to DRAM takes several thousands of cycles. Next we have the caches, which are divided as levels. Typically, the processor of a laptop has two levels of caches: Level 2 or L2 is usually in the range of 128 kilobyte to 2 megabyte, can be implemented on the CPU chip or off, and can take up to 12 cycles to access. Level 1 cache or L1 cache is on the same chip as the CPU. An L1 cache can range between 2 kilobyte and 32 kilobytes can take between 1 and 4 cycles to access. In most processors, there are two level one caches, and instruction level 1 cache that is used for the instruction fetches of the CPU pipeline, and a data level 1 cache that is for the memory stage of the pipeline. The upmost level of the memory hierarchy is the general-purpose registers, which are part of the CPU and take one cycle to access

So with this illustration, we can imagine how instructions are moved across the different levels of the memory hierarchy. When we start an application, its binary and its static data are copied from the hard disk to the main memory. When the application starts executing, code and data are copied from the main memory to the caches. But since the caches are significantly smaller than the main memory, data and instructions have to be moved back and forth between the caches and main memory in order get the correct ones in the cache for execution

DRAM and Cache Blocks 1.6

So now let us look at how caches are organized and how data and instructions are moved from the main memory to the cache. Let us first go over some basics. First, the main memory is organized into blocks, each block being of the same size. The CPU designer can configure the memory blocks to be 16, 32, or 64 bytes. Each cache in the CPU is also organized into blocks that are the same sizes as the main memory blocks. But since the size of a cache is much smaller than that of the main memory, there are significantly less cache blocks than there are main memory blocks. For example, if we have a memory hierarchy with 16-byte blocks, a 2 gigabyte main memory will have over 1 million blocks, whereas a 32 kilobyte cache has a little bit more than 2 thousand blocks. So the key question is how does the CPU move data and instructions between the caches and the main memory?

Memory Hierarchy – Locality Principles 1.6.1 (Temporal Locality)

For that the cache uses two locality principles known as temporal and spatial locality principles of temporal and spatial locality. Temporal locality basically states that if a data or instructions is used once, it is likely to be used again soon. Therefore, once an instruction or data is brought into the cache, temporal locality suggests to keep it the cache as long as possible. One of the best examples of temporal locality is loop code. When we go in a loop once, we are very likely to go through it again. Therefore, it is best to keep the loop code in the cache, rather than overwrite it or move it back in the main memory.

Memory Hierarchy – Locality Principles 1.6.2 (Spatial Locality)

Spatial locality states that if a data or instruction is used, then the instruction or data next to it is also likely to be accessed soon. The clearest example to illustrate spatial locality is with arrays; once you access the elements in the one cell of an array, you are likely to access the elements in the next cells. Using this spatial locality, when a data or instruction is accessed, instead of bringing only data or instruction from the memory to the cache, the whole block that holds this instruction or data is brought in. Consider the example here. We have a code that accumulates the numbers of an array. Let us assume that we have 32-byte cache and DRAM blocks and the elements of the array are 8 bytes. Once we go to the main memory to get eight bytes for the first elements of the array, we bring the 8 bytes for that element as well as the 24 bytes next to it. In this case, if the second, third, and fourth elements of the array are needed, they are already in the cache and are faster to access from there than if we had to go back to the main memory to get them.

Cache Configuration 1.7

So spatial locality is a bit easier to implement. This is because, the main thing to consider with that is the size of the cache block. The larger the cache block, the more data or instructions that are next to each other we can bring and the more spatial locality we have. And since the cache block size is the same as the DRAM block size, there is not much the cache designer can do to impact that. On the other hand, temporal locality requires a bit of work. Ideally, what we would like to do is keep all the D RAM blocks in the cache. But we know that is not practical because of the size differences. So the CPU designer has to make several design choices in the cache in order to maintain temporal locality as much as possible. For that, the designer uses the following parameters: the associativity of the cache to determine which locations in the cache can a D RAM block be moved to, the replacement policy to determine when to evict a block from the cache, and the write policy which determines how to synchronize the data between the cache and the D RAM. We will discuss how each parameter impacts not only the temporal locality but also the delay at which the cache is accessed

Knowledge Check 1.8 (Slide 11)

Answer A

Since a DRAM block can be as large as 64 bytes, and each instruction is only 4 bytes, then a DRAM block can have up to 16 instructions

Answer B

A cache block is the same size as that of a DRAM block. Therefore, a cache block can hold only one DRAM block at a time.

Answer C

The furthest memory from the CPU is the hard disk and it is the biggest size. The closest memory to the CPU is the set of general-purpose registers and it is the smallest size

Answer D

That is correct. The general-purpose registers are the closest to memory and they only need one cycle to access.

Direct Map 1.9

So let us discuss cache associativity. Cache associativity dictates where in the cache can a DRAM block be copied into. Associativity is the main factor that determines the trade-off between temporal locality and access time of the data in the cache. There are generally three associativity methods: one is direct mapped, the other is fully associative, and the last one is set associative

In a direct map cache, there is a one-to-one mapping between the cache blocks and the D RAM blocks. In other words, a D RAM block can be moved to only one specific cache block. We use this color coding to indicate direct mapping. Blue blocks in the D RAM can only go the one blue block in the cache. Yellow blocks in the D RAM can only go the yellow block in the cache, so on and so forth. The main benefit of the direct map cache is it is easier and faster to search the cache. This is because, since a D RAM block can be moved to only one specific cache block, we can directly go to the exact cache block to look for the data. Usually, the way that is done is by using the index number of the cache block. In this example we have here, when searching for the instruction or data in a D RAM block 0 or 4, the cache controller simply goes to cache block 0 to get the instruction or data because that is the only place in the cache that D RAM blue blocks can go to in the cache.

However, direct map doesn't provide good temporal locality. This is because the blue cache block may have to be evicted several times if another blue D RAM block needs to be brought onto the cache. When we say eviction we essentially mean that cache block has to overwritten or replaced with new content coming from the D RAM. Since direct map only allows one location to put all the blue D RAM blocks, if the blue cache block is busy, then we have to kick the current content of the blue cache block out. If we end up needing it again, we have to go back to the D RAM to get, causing more delay. In summary, direct mapping provides very good access speed because it is easy to search, but has bad temporal locality.

Fully Associativity 1.10

The next cache associativity we will discuss is full associativity. In a fully-associative cache, a D RAM block can go into any cache block that is unoccupied. In the figure, the blue D RAM block can go to any cache block that currently doesn't have data or instruction. So the fully-associative cache is the opposite of direct map. It tries to keep the content in the cache for as long as possible by allowing it to be put anywhere that is free. Therefore, fully-associative cache has very good temporal locality. However, since the data or instruction can be anywhere in the cache, we may have to look to the whole cache to find it. If we look at the cache as an array of blocks, it may take several cycles to get the data or instructions, leading to more access delay.

Knowledge Check 1.11 (Slide 19)

Answer A

Remember, in a direct-mapped cache, a DRAM block can only go to one location in the cache. So there is no need to search for the whole cache, one can simply go to that one location. Thus it takes one clock cycle to search the cache

Answer B

That is correct. In a fully associative cache, a DRAM block can go to any location in the cache. Therefore, when searching the cache, we have to search all possible locations to find the data.

Answer C

A direct-mapped cache uses a one-to-one approach, where there is only one location in the cache that a DRAM block can go.

Answer D

A fully-associative cache uses a one-to-any policy, where a DRAM block can go to any available cache block.

Set Associative 1.12

So we see how the two extremes of cache associativity work. What we want to do is to get the best from both extremes. This is where set associativity comes in. In a set associative cache, the cache is broken down into sets of N equal blocks. In the figure we have here, the cache is broken down into sets of 2 blocks. The idea is to make the cache direct-mapped with respect to the sets, but fully associative within a given set. Here is what we mean; each DRAM block can only go to a specific cache set. For example, the D RAM blocks in blue can only go the set 0 of the cache, and all D RAM blocks in yellow can only go cache set 1. In this respect, the cache is direct-mapped with respect to which set the D RAM blocks can go to. However, within a set, a D RAM block can go to any free cache block of that set. For example, with 2 blocks in each set, a blue D RAM block can go to any block that is free in set 0. Similarly, a yellow D RAM block can go to any block in set 1.

So with this hybrid design, the set associativity principle attempts to find the middle ground between direct mapped and fully-associative. Though we use 2 blocks per set in our example, this number can be

different. Usually, a set-associative cache in modern processors has 4 or 8 blocks per set. The general label for a set associative cache is N-way set associative, where N is the number of blocks per set.

Cache Block Replacement Policy 1.13

Let us now discuss the replacement policy of a cache. The goal of replacement policy is to try to keep the block in the cache for as long as possible based on the probability that it will be used again in the near future. This is done to improve temporal locality.

If we have a direct-mapped, then there is not much of an option. Since blue D RAM blocks can only go to only one cache block, if a new blue D RAM block is being brought into the cache, then we have to remove the current content cache block 0. So direct mapping doesn't provide very good options for replacement. This is another reason why it is not ideal for temporal locality.

If we have a fully-associative cache, things are a little bit different. This is because in a fully-associative cache, we have options. Remember, in a fully-associative cache, a D RAM block can move anywhere in the cache. If a block is free in the cache, then we can always move the new D RAM block to that free location. So there is nothing to evict in that current case. Let us consider the worst case that all the blocks of the fully-associative cache are occupied, and we must evict something.

The most common replacement policy is called the least recently used policy, or LRU policy. The idea behind the LRU policy is to evict the cache block that is the oldest to have been accessed. The intuition here is that if the content of a cache have not been accessed recently, then they are unlikely to be accessed again in the near future. So we can evict it. On the other hand, if the contents of a cache block have been accessed recently, they will likely be used again soon. So it is best to keep it in the cache to maintain temporal locality. To determine the recency of use, each cache block has what are called age bits. The age bits of a cache block work as a time clock that indicates the last time a cache block was accessed. Whenever a cache block is accessed, its age bits are updated to represent the current time of accessed.

So in our example, we have a fully associative cache and all cache blocks are occupied. So the way the LRU policy works is that it first reads the age bits of all cache blocks and find cache block with the lowest value age bits. Remember, the least recently used block will have the smallest value in its age bits. This oldest block is chosen for replacement. This way, blocks that have been used recently can stay in the cache because they are more likely to be used again soon according to temporal locality.

So in this example, the least recently used cache block is the last one because its age bits are the smallest when we convert all the age bits to decimal values. So that cache block would be selected for eviction. During the eviction, the content of the new D RAM block is brought into the cache block and the age bits of this block are updated to indicate that it was recently used. As we see, now this cache block is the most recent one to have been accessed in the cache because its age bits have the highest value. The LRU policy also works for set associative caches. But remember, in a set-associative cache, a D

RAM block can only go to a specific set within a cache. So consider the example of a 2-way set associative cache. A new D RAM block needs to be brought up to the cache. Since this D RAM block is blue, it can only go to set 0 of the cache. Since both blocks in that set are occupied, one of them must be evicted. So using the LRU policy, the cache controller reads the age bits of the two cache blocks and notices that the first one is the oldest used. The content of that cache block is thus evicted, the new D RAM block is brought in, and the age bits of the block are updated to reflect the current time.

Although we have only discussed LRU policy, there are other common approaches including least frequently used, which uses counters to keep track how often caches are used, random replacement, which randomly chooses a block to evict, and several hybrid methods that dynamically adjust between least recently used and least frequently used based on the performance of the cache.

Knowledge Check 1.14 (Slide 32)

Answer A

That is correct. In a fully associative cache, the new DRAM block can go anywhere in the cache as long as there is a free block. So regardless of the replacement policy, if there is a cache block that is available, the DRAM block is put in that available cache block. In our example, the 32nd block is free

Answer B

Remember, in a fully associative cache, a cache block is replaced only if all the cache blocks are occupied. If there is a free block, we can put it in there.

Answer C

Remember, in a fully associative cache, a cache block is replaced only if all the cache blocks are occupied. If there is a free block, we can put it in there.

Answer D

Remember, in a fully associative cache, a cache block is replaced only if all the cache blocks are occupied. If there is a free block, we can put it in there.

Cache Write Policy 1.15

The next aspect of cache design we will look at is the write policy. Though cache policy has some impact on temporal locality, it is primarily done to synchronize the data in the cache with that in main memory. For example, when we have a store instruction, do we write the data in the cache only, in the D RAM only, or in both? The decision on which to select depends on the bandwidth limitations of the memory, the amount of energy it takes to write to memory, and who else needs the data. For example, if we have a processor for a low-power device like a smartphone. Its memory is likely to have limited bandwidth, so it may take a lot of time and energy to always write to the D RAM for every store instruction. However, if the data that we are writing needs to be used by a co-processor like a graphics processor, then it is best to have it in D RAM where the co-processor can access it. There are two common write policies: write-through and write-back. The basic idea of write-through is to write the data both to cache and the D RAM on a store instruction. This keeps both the cache and D RAM synchronized but consumes more

energy. In write-back cache, the data is written to the cache only on a store instruction. The data is not written into the D RAM until the cache block has to be evicted according to its replacement policy.

WriteBack Policy 1.16

If it worth emphasizing the write-back policy of a cache and the write-back stage of a CPU pipeline. The write-back policy of a cache deals with synchronizing data between the cache and the D RAM. Whereas, the write-back stage of a CPU is for writing registers at the end of instruction execution. So how is the data in the write-back cache synchronized with the DRAM?

The Write-Back policy synchronizes the DRAM when it has to evict the cache block that a store instruction was performed on. Let us take a look at an example. Assume we have 2-way set associative LRU cache with the write-back policy. Each block of a write-back cache has what we call a dirty bit. The role of the dirty bit is to indicate that a store instruction has been performed on the block. So on a store instruction, the data is written into the cache block only and the dirty bit for that cache block is set to one. In addition, the age bits of the block are updated to reflect the current time. So in our example, the last block of the cache is being written to on the store instruction.

Later in the execution, a new D RAM block needs to be brought into the cache. Since the cache uses a 2-way set associative policy, we go to the set only set that the new DR AM block can be moved to. The new D RAM block is yellow, so it can only go to set 1 in the cache. The cache controller realizes that all blocks in the set are occupied, so one of them must be evicted. Since the cache replacement policy is LRU, the cache controller looks at the age bits of the two blocks in set 1. The last block is the oldest and thus must be evicted. Before evicting it, the cache controller checks its dirty bit. If the dirty bit is one, this means that a store instruction was performed on that block and the DRAM doesn't have the new data. So the cache controller writes the data of that block in the DRAM to updated the DRAM and clears the dirty bit. Then, the new block can be brought in that cache.

In this example here, the content of the dirty cache block is written to the D RAM to synchronize it and the dirty bit for that cache is clear. At this point, the D RAM has the must up to date information for the block. Then the new DRAM block is brought in the recently evicted cache block and its age bits are updated to reflect the current time.

Cache Addressing 1.17

Ok. So we have covered the different parameters for cache organization. Now let us see how exactly the cache is accessed when the CPU gives it an address during the instruction fetch stage or during the memory stage. This figure here shows how the cache controller breaks down a physical memory address into different fields to find where in the cache is the data or instruction associated to that address is. All of these field are in bits. The rightmost field of the address is the cache block offset. We will explain the role of the cache block offset later. The middle field of the address is the cache block number if the cache is direct mapped, or the cache set number if the cache is set associative. Therefore,

this field indicates which cache block number or set block number that the content of this physical address must be in if it in the cache. The last field is the cache, address, is the cache tag. We will explain the role of the cache tag later.

The D RAM block number is the concatenation of the cache tag and either the cache set number or the cache block number, depending on the associativity of the cache. Here we see the different formulas to obtain the size of the different fields. Note that all results of the formulas are in bits. Let us consider an example. Assume that we have a memory hierarchy with 16-byte blocks, a 16-megabyte D RAM and with Level 1 caches. Each Level 1 cache is 16 kilobyte and is 2-way set associative. Based on this configuration, we can calculate use our formulas to calculate the size of each field for the physical addresses. Note that here when we say physical address what we mean the location of the data or instruction in D RAM. Also note that it is simpler to calculate the size of the fields by using the two to the power notation. For example, 16 megabyte is 2 to the power of 24, therefore, the address is 24 bits.

Cache Addressing 1.18

Now let us now see the role of the block offset and the cache tag fields in the physical address. Remember that each D RAM or cache block is at least 16 bytes. In MIPS64, an instruction is four bytes and an instruction in. So each block can have four instructions. The role of the cache block offset field is to indicate which of the four instructions in the block to access for a given address. In case of a data access, each data can be at most 8 bytes in MIPS64. So a cache block of 16 bytes can up to two double words. So the goal of the offset is to indicate which of the two double words the address is requesting

Let us see the role of the cache tag. Assume we have a 16 megabyte D RAM and a 16 kilobyte cache with a 2-way set associative. Now consider the following addresses: 001C and 004018. When we break the addresses into the different fields, we see that they both have same cache set 0. This means, when accessing these addresses, the cache controller will go to same cache set 1 to find. However, they have different D RAM blocks, which makes sense because they are different addresses. So, the role of the cache tag is to determine which D RAM block that the content of a cache block belongs to. So, if we concatenate the cache tag and the cache set, we get the D RAM block. So, when searching for a data in the cache, the cache controller has to also compare the cache tags to see if it is getting the correct instruction or data. We will see an illustration of this later.

Cache Usage During Execution 1.19

So let us take a look at how the cache is accessed during the instruction fetch and the memory stages of the CPU pipeline. Let us assume we have 16 megabyte DRAM, and a 16 kilobye, 2-way set associative cache with LRU replacement and WriteBack policy. In addition to the cache age bits and the dirty bits that we discussed before, each cache block has a valid bit and cache tag bits. The valid bit indicates that there is a valid content in the cache block. So initially the valid bit is zero and when something is brought into the cache the first time, the valid bit for that cache block is set to one. The cache tag bits hold the value for the cache tag field for the D RAM address of the data or instruction in that cache block. We will illustrate how the cache controller goes about searching the cache and updating it given an address. By default, when the CPU starts, all the valid bits are zero because there is nothing in the cache. This is

known as a cold start. For simplicity, we assume this is the L1 data cache. So it only handles requests for data accesses at the memory stage of the pipeline

At some point in execution, there is a Load double instruction where the memory address is 001C. So the CPU calculates the target address and gives it to the cache controller. The cache controller then breaks down the address into the different fields to find out if the requested data is in the cache. Here we see that the cache controller gets the fields and determines that if the data for this load double instruction is already in the cache, then it must be in set 1. The cache controller searches for each block in set 1. But since all the valid bits of the blocks in that set are all zero, then there is no data in that cache set. Hence the data is not in the cache and must be brought from the D RAM. This is known as a cache miss. Though we are using the LRU policy, there is nothing to replace because the set is initially empty. So the cache controller simply sends the address to the memory controller to handle the cache miss. The memory controller uses the D RAM block field of the address to find out which block is being requested. Here we see that D RAM block is requested. So the memory controller gets D RAM block 1 and sends it back to the cache controller.

The cache controller then takes the data, adds it to an empty blocks in the set, updates the valid bit to indicate that there is some valid content in that cache block, updates the cache age bits to indicate the recency of access for this block and sets the cache tag bits of that block to the value of the cache tag field of the address. In our case, the cache tag field is zero.

Cache Usage During Execution 1.20

Later in execution, consider the cache looks like this. In addition, we get a new load double instruction where the address now is 0014. The CPU sends the address to the cache controller and the latter breaks it down into the different fields. The controller calculates the set of this address and it realizes it's one. The cache controller then begins searching for the cache blocks in set 1. For each block in the set, the controller first check the valid bit of that block. If the bit is not one, then there is no valid data, so there is no need to continue with that block. If the valid bit is one, the controller then compares the cache tag field of the requested address to the cache tag bits for that block. If the comparison matches, then the cache block has the data we are looking for. This is because when we combine the cache tag and the cache set, we get the D RAM block. Therefore, if they both match, then the content of the requested D RAM block is in that matching cache block. This is known as a cache hit. When there is a cache hit, the data is brought from the cache to CPU, without the need to go to memory. Hence we get much faster access of the data. Cache hit and cache miss ratios are the main metric of performance of a cache. The goal is to maximize the percentage of hits that we have for all memory accesses that the CPU makes.

Topics Covered 1.21

We have covered memory hierarchy. We discussed the main limitations of D RAM with respect to access delay and how it can have a significant degradation on performance. To overcome this delay, we discuss how caches can be used. We highlighted the principles of temporal and spatial locality of caches and how they can be organized to keep frequently used data and instructions closer to the CPU in order to reduce access delay.