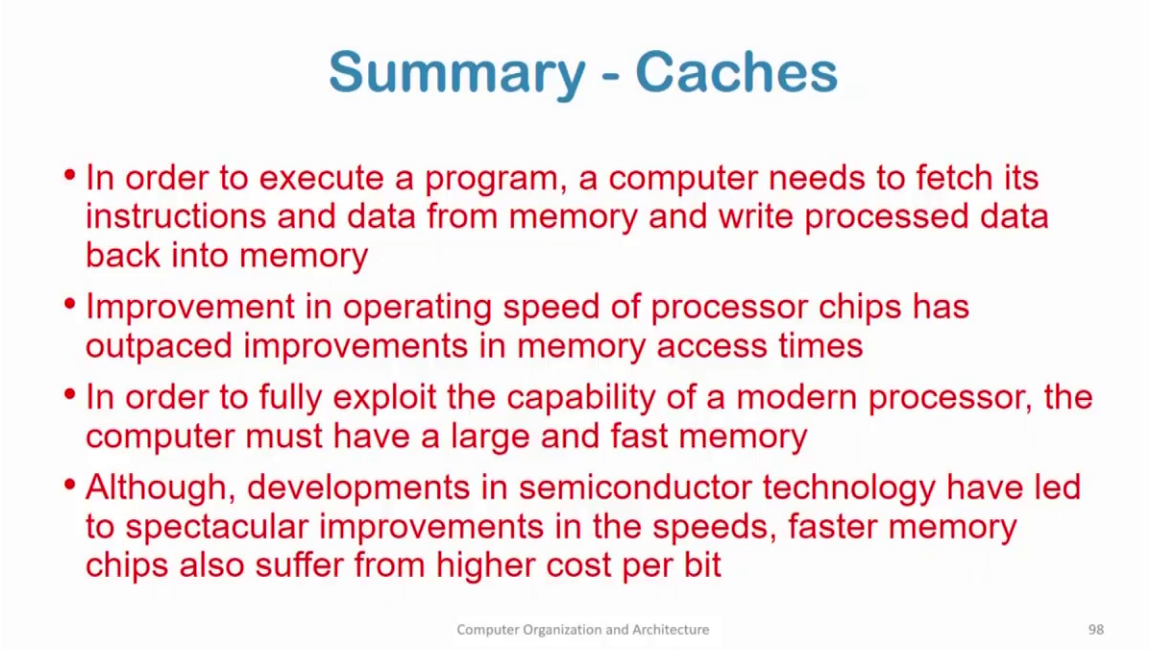
Computer Organization and Architecture: A Pedagogical Aspect

Prof. Jatindra Kr. Deka
Dr. Santosh Biswas
Dr. Arnab Sarkar
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture – 28
Summary

(Refer Slide Time: 00:26)



This is a small lecture where we summarize our discussion in the last 3 lectures. So, in order to execute a program a computer needs to fetch its instructions and data from memory and write process data back into memory. So, in each instruction you need memory from where the instructions, from which the instructions will be fetched, and data also needs to be data needed by the program needs to be fetched and also written back to. Improvement in operating speed of processor chips has outpaced improvements in memory access times.

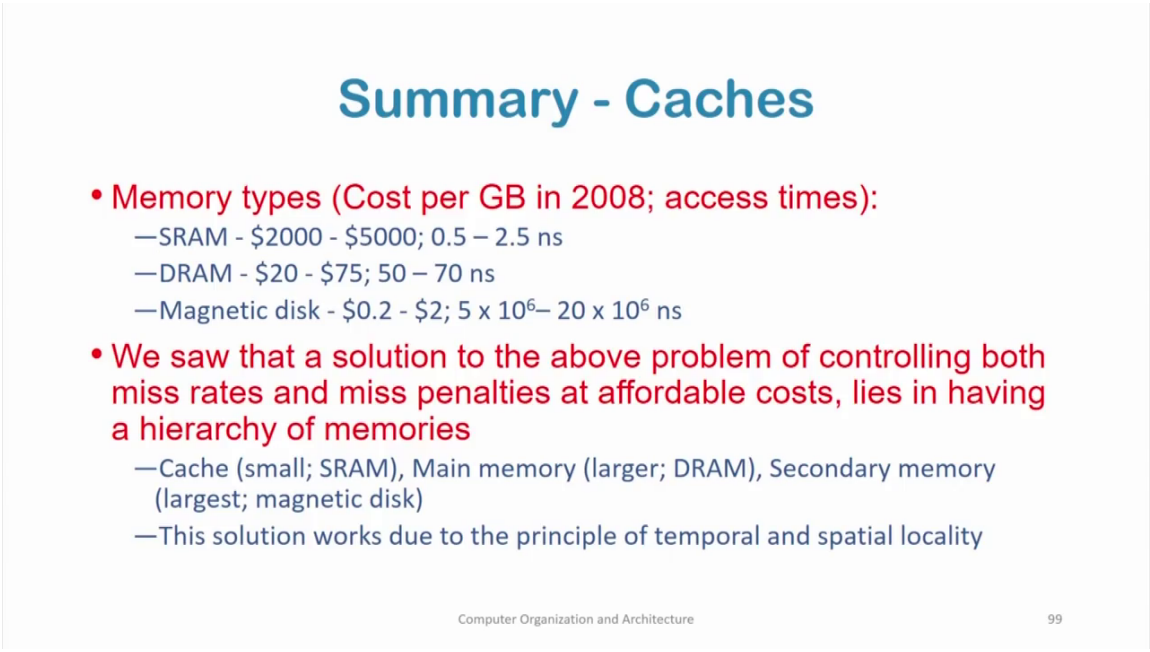
Now, so therefore, we see that for each instruction we need memory. Now the first problem is that the speed of processor chips has outpaced improvements in memory access times. So, the rate the speed at which the processor can execute instructions it is much faster than improvements in memory access time than the speed at which memory

can be accessed. So, effectively the instructions because each instruction needs memory if some mechanism is not adapted and we have the slow memory, the execution of instructions will be as fast as the speed at which memory can be accessed.

Now, the other problem is that too fully exploit the capability of a model processor, the computer must have large and fast memory. So, we have lots of programs and all those programs may execute simultaneously on the processor may be co executing on the processor at a given time. And all these programs therefore need to be in the memory. Now to keep all these programs in memory and individual programs are also growing in size are very big programs. So, you need therefore, very big memories to keep all these programs in memory and allow them to co execute at the same time on the processor. Although, developments in semiconductor technology have led to spectacular improvements in the speeds, faster memory chips also suffer from higher cost per bit.

So, although now we have improvements in speeds or access times, improvements in access times of memory fast memories also have higher cost per bit.

(Refer Slide Time: 03:05)



Now, given different memory types we have various technologies by which memories can be developed. And here for example, SRAM, DRAM and magnetic disk has been shown. Their cost per GB as of in 2008 and their access times were as follows. For SRAM the cost per GB of memory was 2000 to 5000 dollars, which is very high for

every GB of cache SRAM memory. And their access times although were very fast 0.5 to 2.5 nanoseconds.

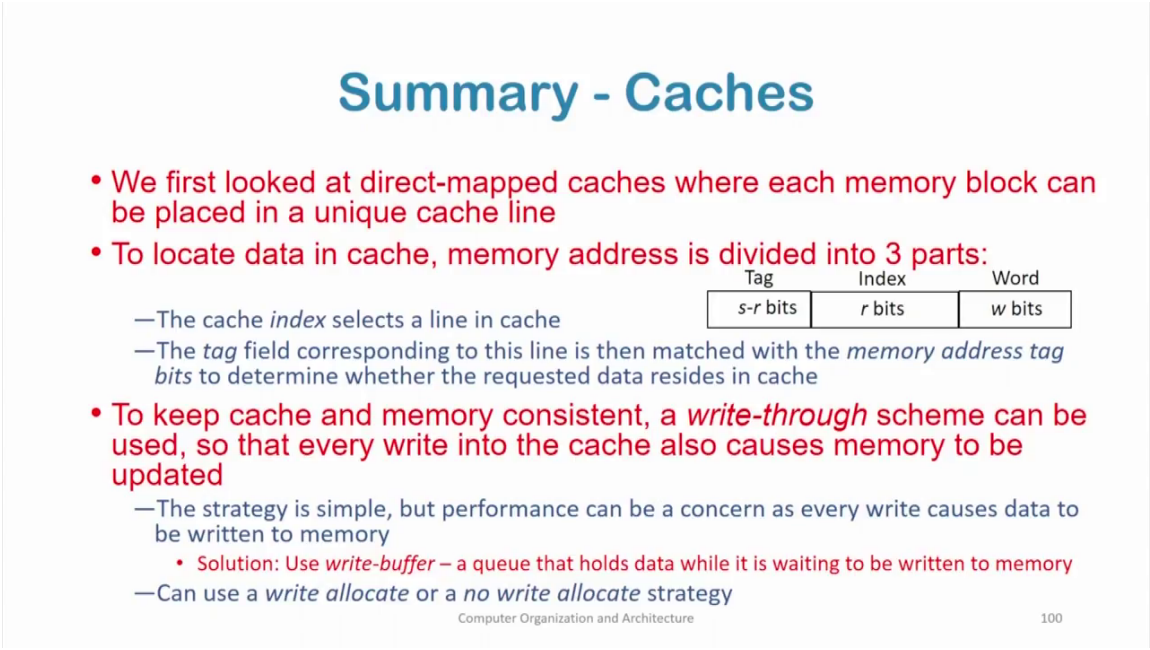
So, access times are almost as fast as processor speeds. Now DRAM which is mainly used for the main memory, their cost per GB is much lower, only 20 dollars to 75 dollars compared to SRAM's; which is 2000 to 5000 dollars, but their access times are also much larger than the access times of SRAM. They are 50 to 70 nanoseconds. For magnetic tapes, the cost per GB is still much lower it is 0.2 to 2.5 dollars per GB, but the access times is also much much higher than main memory.

Now, because we need a very large memory, but we saw that fast memories, we cannot have very fast large memories, because the cost of these fast memories are also very huge. So, we need to have a tradeoff between the cost and the size of memory that is needed. And we saw that a solution to the above problem of controlling both miss rates and miss penalties at affordable costs lies in having a hierarchy of memories.

So, we have a small cache built using SRAM, larger main memory built using DRAM, and even larger secondary memory built using magnetic disk. And then we saw that this solution that having this hierarchical memory works this solution works because of the principle of temporal and spatial locality.

Where, the principle of temporal locality says that data items that are accessed currently is expected to be has a high probability of being accessed in the near future. Data items which I am accessing now will be accessed again in the near future, and the temporal of spatial locality which says that data in the memory in the vicinity of those which are being accessed currently is expected to be accessed soon. So, data which are near to the data that I am accessing is expected to be accessed soon.

(Refer Slide Time: 05:59)



When we started talking about caches in particular, we first looked at direct mapped caches where each memory block can be placed in a unique cache line. So, while talking of caches we said that we divide the memory as well as the caches into same sized blocks.

So, the divisions in memory will be called blocks the divisions in caches will be called lines although the definition is not exactly hard and fast, we often tend to use blocks for caches as well. But, in general we use cache lines we tell cache lines and memory blocks. To locate data in cache memory address is divided into 3 parts. So, each physical each memory address generated by the processor is divided into 3 parts.

The tag bits, the index bits and the word offset; where the word offsets which we are showing here as w bits, with the way the word offset which we are showing is w bits is used to identify a particular word within a block. The index of r bits is used to access a particular block within the cache, and the tag bits are used to match if a particular block of memory is in cache or not.

So, the cache index selects a line in cache, the tag field corresponding to this line is then matched. So, I have a line in cache, and this line is identified by the index bits. And in that line I have a tag field, and that tag field is then matched with the memory address tag to tag bits to determine whether the requested data resides in cache. To keep cache and

memory consistent a right through scheme can be used so that every right into the cache causes memory to be updated.

So, in a write through cache whenever I write on to the cache I also write on to the main memory. The strategy is simple, but performance can be a concern as every write causes data to be written to memory. So, whenever I write on to cache I have to access the main memory which is much slower. So, although the strategy is simple, because when I need to replace a cache, I don't need to see whether the data in cache is dirty whether the data in cache needs to be written to memory before replacement.

It's always consistent with the memory. And therefore, the during replacement a cache line can simply be discarded and a new block of memory can be fit into this cache line. So, the solution to this performance problem of write through caches is to use a write buffer a queue. So, what is the write buffer a queue that holds data while it is writing while it is waiting to be to be written to memory ok. So, then what is a write buffer? So, I have to whatever I am writing in cache I have to write in memory.

Now when I am using a write buffer, I write to cache and I write into the write buffer. And this write buffer then subsequently writes it to the memory. After writing into the write buffer the processor is then free to go with the next instruction, it does not wait for the cache for this data to be written to memory before starting the next instruction.

The processor can go about it is all work crunching the next instruction okay by using a write buffer. Now write, write misses you know when I am when we are talking about writes in a write through cache we can use a write allocate or a know write allocate policy.

In a write allocate policy when we have a write miss; so, I am trying to write on to the cache and the particular line and the particular block of memory in which I have to write is not currently there in cache. So, in this situation in a write allocate policy I first bring the block of memory from the main memory into the cache and then write on to this cache line. And write on to this cache line subsequently I also write into the memory because it's a write through cache.

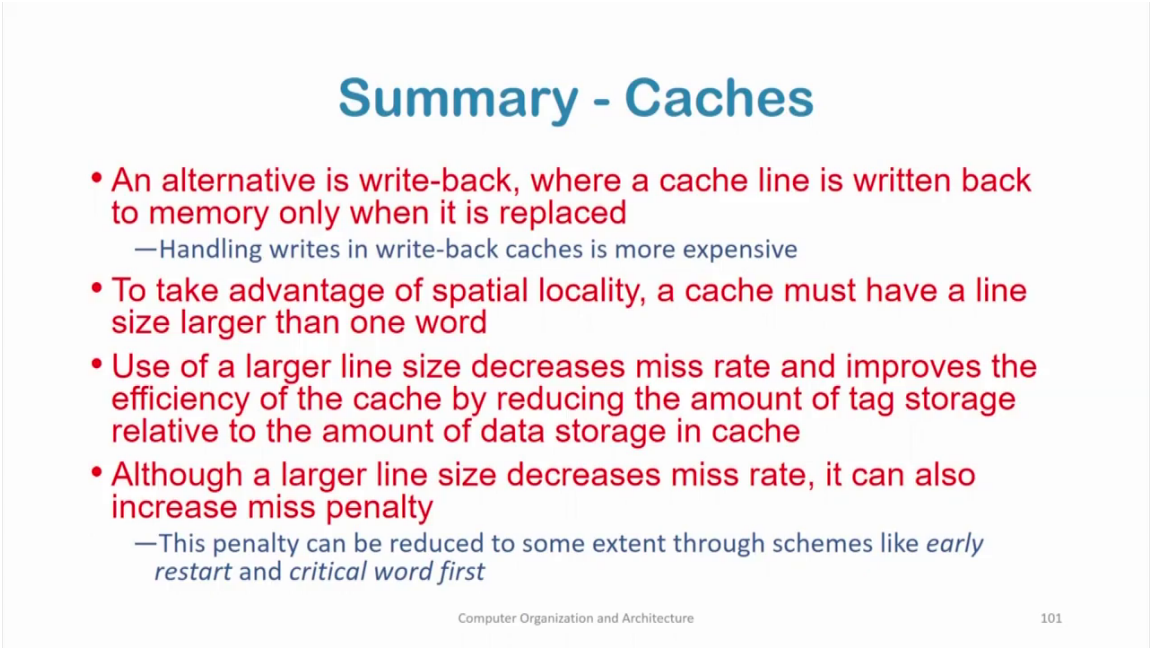
Now, in a no write allocate policy, when I have a write miss, and I have to and when I have a write miss. That means, the block of memory is not in cache, I do not just allocate

a line in cache for this memory block, I directly write on to the cache sorry, I directly write on to the main memory. I don't write in to the cache at all. So, in a no write allocate policy, what is this policy? I have we are talking about write misses in a write through cache; whenever there is a write miss in a write through cache, and I am using a no write allocate policy, I don't write into the cache at all.

I directly write into the main memory. One advantage is for repetitive writes for example, I am initializing a large block of memory say, I go on writing on to the memory, I don't have to allocate bring this block of memory into the cache and then write ok.

So, this gives a good performance gain.

(Refer Slide Time: 11:29)



An alternative to write through is write back; where a cache line is written back to memory only when it is replaced. A cache line is written back into memory only when it is replaced. So now, when I am writing on to I am writing on to the cache, I don't write into the memory. If that cache, if that cache line is if that memory block is in cache, I only write into the cache I don't write into the memory.

So, repetitive writes don't onto the same block, we need not new repetitively write each write into the cache into the memory in update I we don't need to update that. So, if I

have multiple writes onto the same block, then I can take all these writes together the modified dirty cache, and when being replaced I can just write it once into memory.

So, this is the advantage. However, handling writes in a write back cache is more expensive. To take advantage of spatial locality a cache must have larger size than one word. So, what did spatial locality said? It said that the data that is accessed. So, if I have a data that is accessed, the data nearby this data in memory is expected to be accessed soon. Now to take advantage of that, I need a cache whose line size is larger than one word.

If it was one word, then I have not brought words in it's vicinity onto the cache, and therefore, even if the next word is accessed again that has not been fetched and that next word falls in the same. Because it is just one word, the blocks are of one-word size, let us say that one word is brought from memory. And find that is accessed and the next word is not brought from memory along with this because the block size is one word. And due to the spatial locality of reference the next word maybe has high probability of being accessed soon.

But even if it is accessed there will be a cache miss because the block size is just one word. To take advantage of spatial locality therefore, a cache must have a line size larger than one word. Use of a larger line size decreases miss rate as we understand why and improves the efficiency of the cache by reducing the amount of tag storage relative to the amount of data storage in cache. Now when we have when we each word is a block, we understand that when I am bringing a data corresponding to each line in cache I have to have the tag field. So, I need to have tag storage separate tag storage for the cache.

Now, this tag is required one tag is required for each block or each line of cache one tag is required for each line of cache. If every word is a block, or every word is a line, then I will require a tag for every word ok. Now if the when the line size becomes larger therefore, the number of tags required is also reduces.

So, for a given size of the cache, there is a reduction when the when the line size increases there is a reduction in the amount of tag storage relative to the amount of data storage in cache. Although a larger line size decreases miss rate, it can also increase miss penalty. Now when the line size a larger line size decreases miss rate as we saw because it improves spatial locality. However, it can increase miss penalty, because now for

every transfer from main memory I have to bring the entire block entire memory block into this line, for every miss that we have.

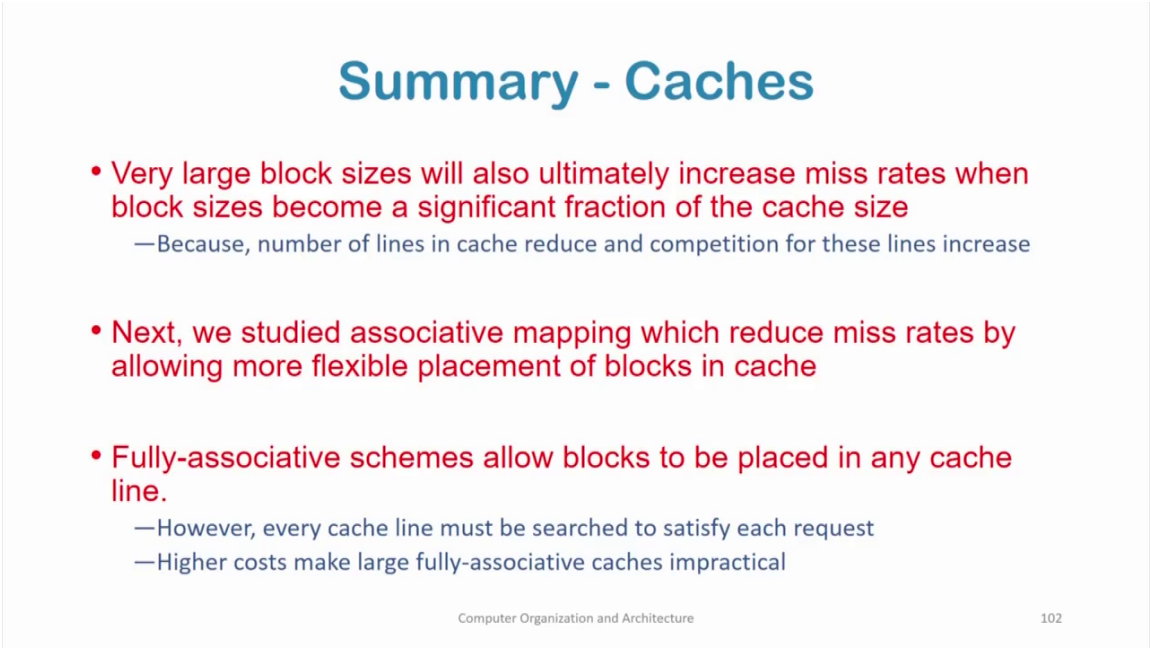
Now when the line size increases, this transfer of block from memory to cache becomes more expensive. This penalty however, can be reduced to some extent through schemes like early restart and critical word first. We saw that in early restart what happens. What happens? When I am fetching from memory a block into cache, I will start the next instruction whenever the required word within that memory block has been fetched.

I don't wait the processor doesn't wait for the entire block to be brought into cache and after which the next instruction can start this doesn't occur. So, in early restart what happens is that I have initiated block transfer and within this block somewhere my requested data word it lies. Now whenever I get my requested data word in cache I start my next instruction, I don't wait for the don't wait for the rest of the block to be fetched from memory into cache.

Now this is how we can save time, if let us say the next instruction also does not require the next instruction does not incur a miss into the cache again. So, then if this does not happen, then the memory transfer can go take it's own time memory transfer of block from of that block into cache can take it is own time, and in parallel the processor can go on executing instructions.

Even a better strategy is was critical is critical word first. In critical word first what happens? Instead of starting the fetch of the block, from the start of the block we start the fetch of the block from the requested memory word that is needed by the processor at this point in time. Now that critical word which is requested by the processor is fetched first, and subsequently the rest of the block is fetched, but because the processor can start once, the critical word has been brought into cache this becomes even faster than early restart.

(Refer Slide Time: 17:36)

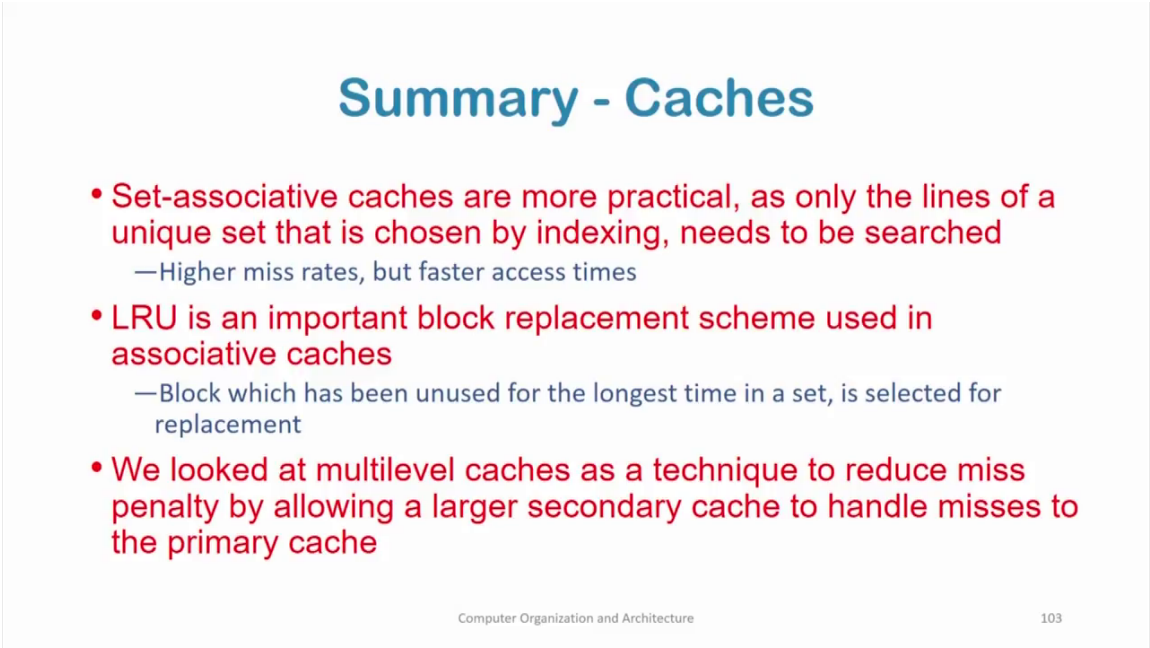


However, very large block sizes will also ultimately increase miss rates when block sizes become a significant fraction of the cache size. Now if you have very large block sizes which are of which are a significant fraction of the cache size, then there will be misses incurred due to insufficient capacity of the cache. Because many of the blocks will because the blocks will compete, because there will be only a few lines in cache and the and the there will be a large very large competition for these few lines in cache.

So, large block sizes will also ultimately increase miss rates, because number of lines in cache reduce and competition for this lines increase. Next we studied associative mapping which reduce miss rates by allowing more flexible placement of blocks in cache. Within this associative mapping we first learnt fully associative schemes which allow blocks to be placed anywhere in cache.

So, memory block can be placed in any line within the cache; however, to have this every cache line must be searched to satisfy each request. Because a memory block can reside in any line in cache I have to search the entire cache to find out whether I have a cache hit or a cache miss and this is expensive. So, we this will have higher costs and higher cost and this such and such higher cost make large fully associative caches impractical.

(Refer Slide Time: 19:16)

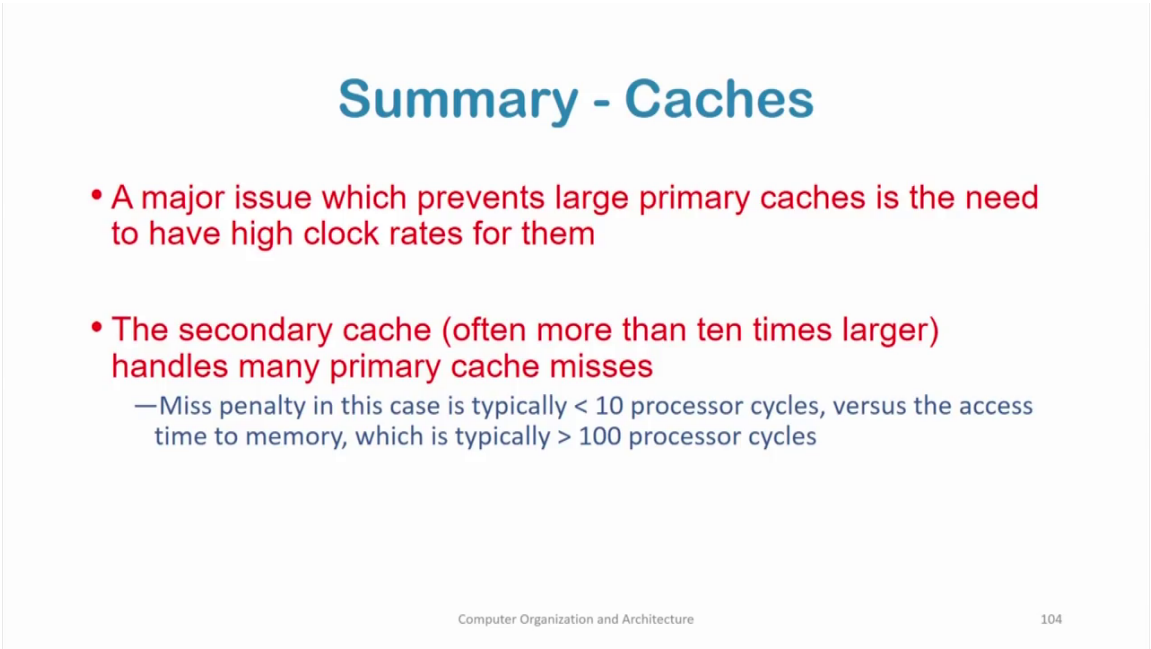


Set associative set associative caches are more practical with respect to fully associative caches, as only the lines of a unique set that is chosen by indexing needs to be searched. So, I have only a few options for a memory block. Not all lines can I cannot put my memory block into any line in cache, I can put my memory block only within a few designated lines which fall within the same set okay.

Higher miss rates than fully as so this scheme incurs higher miss rates than fully associative caches; obviously, because my options are more restricted with respect to fully associative caches. However, I have faster access times, because the search times are much reduced. LRU least recently used is an important block replacement scheme used in associative caches. So, if a block has to be if a line has to be replaced if a line in cache has to be replaced. That means, the block that is currently reside in residing in a set has to be replaced which block should I choose for replacement is the question.

So, block which has been unused for the largest time in a set is selected for replacement ok, you know LRU replacement policy. Then we looked at multi-level caches as a technique to reduce miss penalty by allowing a larger secondary cache to handle misses to the primary cache. Now the so multi-level caches what do we have? We have very small primary caches and then larger and larger secondary caches. So, the first level primary cache is small, it has very fast access times. The second level cache is a is larger, but slower and the third level or other levels are again larger and slower than the primary cache.

(Refer Slide Time: 21:19)



So, a major issue which prevents large primary caches is the need to have a high clock rates for them. So, primary caches are used directly by the instructions by the processor.

So, it has to be as fast as the processor. So, therefore, you cannot have large primary caches, and right the secondary cache often more than 10 times. So, with therefore, we have the secondary caches which are much larger secondary caches often more than 10 times larger handles primary cache misses, now because the primary caches are very small.

There will be misses, there will be more than the probability of misses in the primary cache will be high, and such misses will be handled by the secondary cache which is 10 times or more larger. So, miss penalty in this case is typically less than 10 processor cycles. So, even if you have a miss in the primary cache, the miss penalty is around 10 processor cycles or even lesser into the secondary caches, versus the access to memory. So, even if you did not have a secondary cache, the alternative was that we had to directly access memory which is typically which takes times typically of the order of more than 100 processor cycles. So, this is how why we have advantage by using multi-level caches.

So, this was the summary of our discussions in the first 3 lectures on caches.