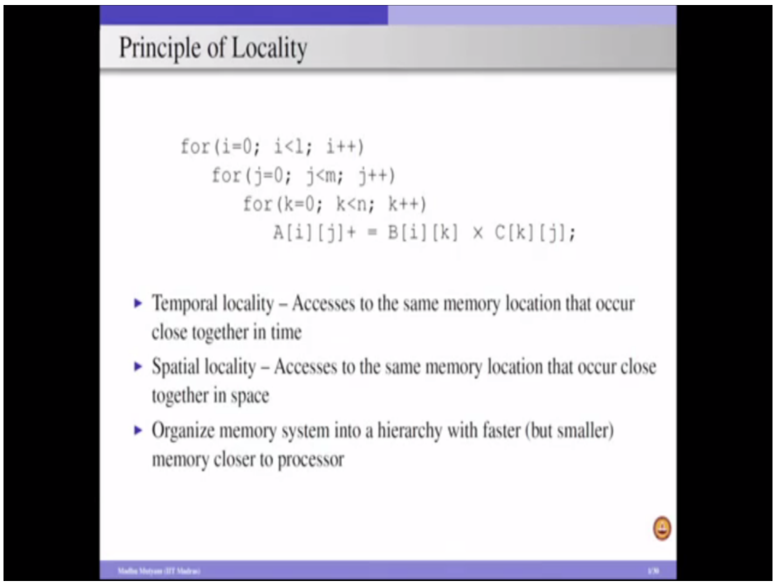
Computer Architecture
Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras.

Module – 03
Lecture - 06
Memory Hierarchy Design (Part 1)

So, the last module we discussed instruction set architecture. And now next 2 weeks we are going to discuss memory hierarchy design mainly consist of cache memory design and the DRAM based main memory. Why do we require to study memory hierarchy design?

We know that programs exhibit principle of locality, which states that the processor accesses some data and instructions now, there is a high chance that the same data will be required in future or the neighboring data may be required in future. So, for example, consider a scenario where the simple piece of code which is matrix multiplication. Here we are performing matrix multiplication on 2 matrices B and C and the result is stored in matrix A

(Refer Slide Time: 01:08)



Principle of Locality

```
for(i=0; i<l; i++)
   for(j=0; j<m; j++)
      for(k=0; k<n; k++)
         A[i][j]+ = B[i][k] × C[k][j];
```

- Temporal locality – Accesses to the same memory location that occur close together in time
- Spatial locality – Accesses to the same memory location that occur close together in space
- Organize memory system into a hierarchy with faster (but smaller) memory closer to processor

We assume in this particular example the data stored in row major order. So, all the elements of a row is stored first and then go the next row and so on. So, given this example we know that B[0][0] is accessed first and then B[0][1], B[0][2] and so on because the k value is changing based on the innermost for loop and i value is fixed for all the iterations of the two inner for loops. So which says that if I access element B[0][0] I am going to access B[0][1] in the next cycle and similarly B[0][2], B[0][3] and so on in the following cycles.

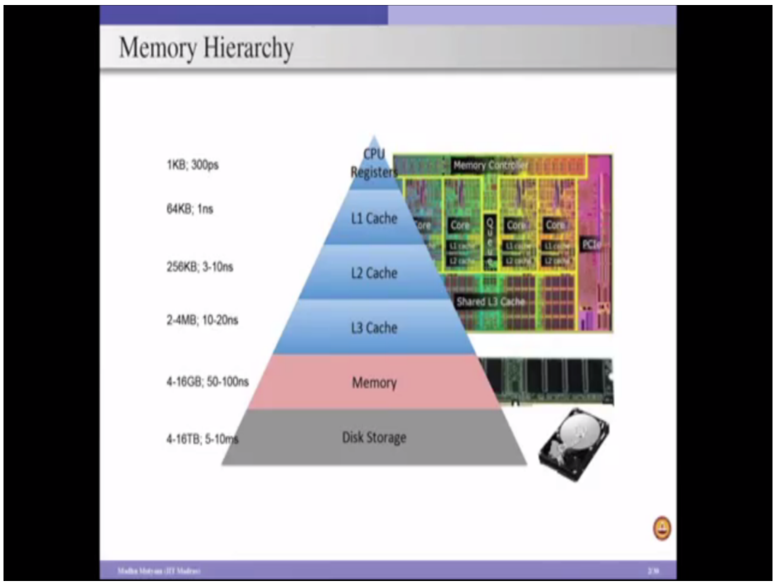
And similarly, when I consider elements of array A, so A[0][0] is repeatedly used for all the iterations of the innermost for loop. That means for k equal to 0 to n, A[0][0] is used and now A[1][0] will be used only after the complete iterations of the innermost for loop is done. And also the complete iterations of the second for loop. So, in the second iteration, so when I equal to 1, again for j equal to 0 to n and k equal to 0 to n. So, we are going to use A[1][0] and after that again this will be repeated for A[1][1], A[1][2] and so on. So, this says that an element of array A when I am accessing now, I am going to access this in the near future. This is true with the other elements of the other arrays also.

From this example it is clear that the elements which are accessed now there is a high chance that, the same elements may be repeatedly used in the near future or elements which are neighboring to the previously accessed elements will be required in the near future. So, this says that the principle of locality can be exploited either in time or in space. So, accesses to the same memory location that occur close together in time is called as temporal locality. Whereas, if it happens close in space that is nothing but accesses to the same memory location that occurs close together in space is called a spatial locality.

And the thumb rule says that ninety percentage of the execution of programs spends in only 10% of the code. So, to exploit this principle of locality available in most of the programs, we need to come up with a memory hierarchy. Rather than storing the entire code and accessing from the memory, it is always better to keep the repeatedly accessed data in the faster memory which is very close to the processor, so that the access time will be very short.

So, effectively all the cache memory designs or the memory hierarchy design what we consider typically exploits the principle of locality that is available in the programs. So, organize your memory system into a hierarchy with the faster, but the smaller memory closer to the processor and large memory which has high access latency will be kept very far from the processor. Let us look at the memory hierarchy.

(Refer Slide Time: 04:54)



Memory Hierarchy

It consists of multiple levels in the hierarchy and this particular example is typically used for server type of systems. At the top of the pyramid we have the processor registers or CPU registers, which typically take less space in the order of less than 1kb, kilobytes of space, but the access time is very fast. It will take typically the orders of picoseconds. The next level of the memory in the memory hierarchy is the level 1 cache memory which again the size of the L1 cache is typically around 32kb to 64kb and the access time it takes 1 nanosecond or something.

As we move further down in the pyramid, the size of the memory is increasing. The access time also is increasing, but the cost per bit is reducing significantly. The size of L2 memory is 256kb to 512kb, depending on the system we consider. And the access times are 3 to 10 nanoseconds and some systems can consider third level of cache which is in the orders of 2 to 4 megabytes of space and takes 10 to 20 nanoseconds. Remember the CPU registers, L1, L2, L3 caches all these are actually designed using SRAM technology, because of that the access times are very low compared to the main memory which is typically designed using DRAM based technology.

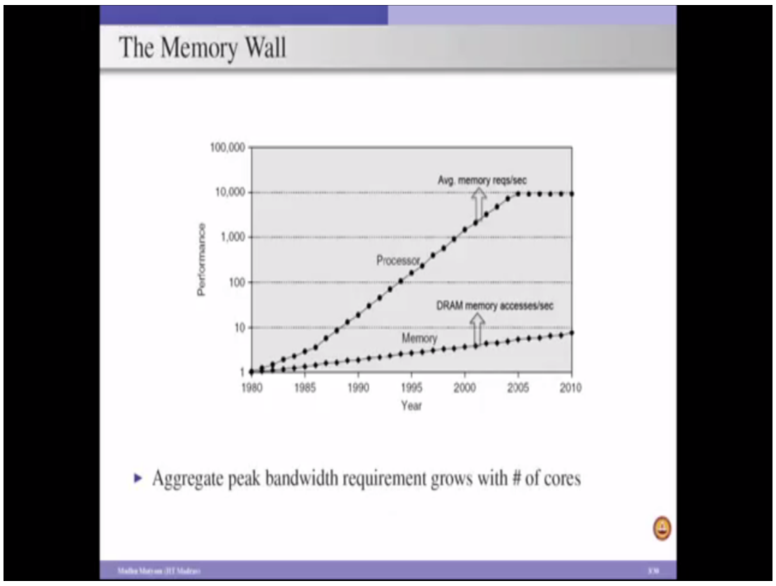
In the case of main memory, as I mentioned earlier we consider DRAM based technology, where a single bit is stored in a DRAM cell which consists of a transistor and a capacitor. And the access times for the main memory is typically in the range of 50 to 100 nanoseconds, but the size of the DRAM based memory is 4 to 16 GB which is very huge.

In the pyramid we have a disc storage, which is having a size in the range of 4 to 16 terabytes, but the access times are in the range of milliseconds mainly because of these mechanical components involved in hard disk. Of course, these days we are also having the solid state drives which are based on flash technology and provide non-volatility. We know that the caches and the memory are volatile memories. So, that means when the power is off, the data will be lost.

We consider flash based memories as a replacement for the hard disc and that is the reason why the latest laptops and so on we are having the flash based solid state drives, but the size of flash based solid state drives is not in the range of the size of HDDs, but they are decent enough to provide good storage space. And the advantage with the solid state drives is their access times are much smaller compared to the HDDs. So, once you have your memory in the multi-levels of hierarchy with the faster memory faster and the smaller sized memory closer to the processor.

So we are going to keep repeatedly accessed data in these faster memories, but, because the size is very small so, we have to be selective in keeping data in these faster memories. So, as part of our cache memory designs we are going to discuss different mechanisms of how to keep selective data. So, is it really required to have so many levels of memory hierarchy in our computer design? Yes it is required this is mainly because of the famous memory wall problem.

(Refer Slide Time: 09:40)



The Memory Wall

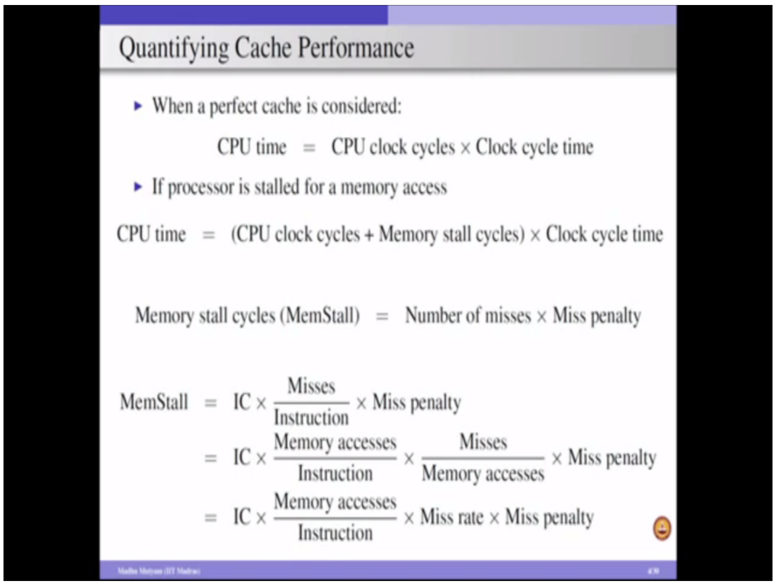
- Aggregate peak bandwidth requirement grows with # of cores

This graph shows that the processor performance is significantly increasing over the years, but the performance of the memory is not increasing at that particular pace. So, as a result as time progresses, we have a significant gap between the processor performance and the DRAM performance. In this graph the X axis shows the time line and the Y axis shows the performance. And the processor line is actually specifies the rate at which the average memory requests per second is increasing and the memory line shows the number of DRAM memory accesses per second.

As we can clearly see that the gap is widening significantly and in order to deal with this wide gap in the performance between the processors and the memory we definitely need to have multi levels of cache hierarchy. And remember this graph is only for a single core processor performance, but these days we are having multi-core systems where two, four, eight, sixteen processors are there in a single chip. And all these cores when they are executing different applications require so much memory band width and to support that there is a significant pressure on the memory.

And so as a result we need to definitely have large spaces of the intermediate memories in the memory hierarchy. So, this says that the aggregate band width requirement grows with the number of cores. So, as a result we need to have efficient memory system hierarchy design which takes care of the demands from the multiple cored of a chip multi-processor so that, the overall performance of the system can be improved. So, having discussed this memory hierarchy and before going to discuss the internals of the cache memory and DRAM memory and so on. Let us see what is the performance improvement we get if the cache memory is provided in the system? To do that, we need to have quantification for cache performance.

(Refer Slide Time: 11:59)



Quantifying Cache Performance

- When a perfect cache is considered:

$$\text{CPU time} = \text{CPU clock cycles} \times \text{Clock cycle time}$$

- If processor is stalled for a memory access

$$\text{CPU time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\text{Memory stall cycles (MemStall)} = \text{Number of misses} \times \text{Miss penalty}$$

$$\begin{aligned}\text{MemStall} &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \\ &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \frac{\text{Misses}}{\text{Memory accesses}} \times \text{Miss penalty} \\ &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\end{aligned}$$

We already discussed the performance equation for CPUs where,

$$CPU\ time = CPU\ clock\ cycles * Clock\ cycle\ time$$

So, here this expression is considered with the assumption that our cache memory is perfect. When I say cache memory is perfect, whatever the memory request issued by the processor it will be satisfied by the cache memory. So, effectively the processor is not stalled for servicing any memory request. This is an ideal scenario, but in reality this is not the case, where processor may be stalled for servicing memory request. So, when you have memory stalls this equation can be written as,

$$CPU\ time = (CPU\ clock\ cycles + Memory\ stall\ cycles) * Clock\ cycle\ time$$

So, CPU clock cycles are the cycles spent by the processor in performing ALU operations or all the requests which are hit in the cache memory. And the memory stall cycles are the cycles the processor sitting idle to get the data from either the 1 level of the cache or 2 levels of the cache or any level in the memory hierarchy. So, memory stall cycles is nothing but ‘the number of misses incurred by the processor’ times ‘the total time incurs for servicing 1 memory miss or 1 cache miss’. So, it is effectively the number of misses times the miss penalty.

$$Memory\ stall\ cycles = Instruction\ count * \frac{Number\ of\ misses}{instruction} * Miss\ penalty$$

Because previously we have given the expression for CPU time in terms of instruction count. So, effectively we can rewrite our memory stalls also in terms of instruction count.

$$Memory\ Stall\ Cycles = Total\ number\ of\ Instructions \text{ in } program * \frac{The\ number\ of\ misses}{instruction} * A\ miss\ penalty$$

This can be further rewritten as,

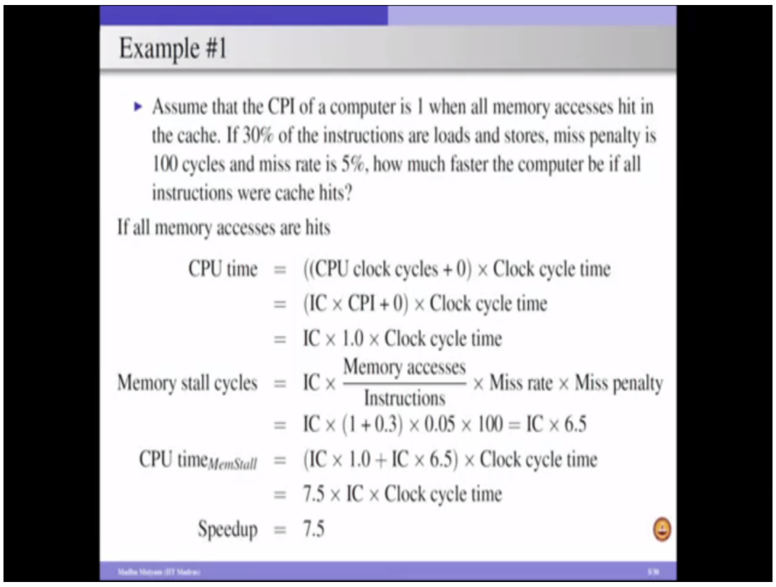
$$Memory\ Stall\ Cycles = Instruction\ count * \frac{Memory\ accesses}{instruction} * \frac{Misses}{memory\ access} * Miss\ penalty$$

So, the ratio of misses per memory accesses, the ratio of misses and memory accesses is called as a miss rate. So, our total memory stalls can be expressed as,

$$Memory\ Stall\ Cycles = Instruction\ count * \frac{Memory\ accesses}{Instruction} * Miss\ rate * Miss\ penalty$$

So, we can substitute this memory stall equation in our CPU time equation to get the overall CPU time if we have a cache memory. And using this equation we can get the performance improvement using a cache memory. So, to illustrate that let us consider an example.

(Refer Slide Time: 15:24)



Example #1

- Assume that the CPI of a computer is 1 when all memory accesses hit in the cache. If 30% of the instructions are loads and stores, miss penalty is 100 cycles and miss rate is 5%, how much faster the computer be if all instructions were cache hits?

If all memory accesses are hits

$$\begin{aligned}
\text{CPU time} &= ((\text{CPU clock cycles} + 0) \times \text{Clock cycle time} \\
&= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle time} \\
&= \text{IC} \times 1.0 \times \text{Clock cycle time} \\
\text{Memory stall cycles} &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instructions}} \times \text{Miss rate} \times \text{Miss penalty} \\
&= \text{IC} \times (1 + 0.3) \times 0.05 \times 100 = \text{IC} \times 6.5 \\
\text{CPU time}_{MemStall} &= (\text{IC} \times 1.0 + \text{IC} \times 6.5) \times \text{Clock cycle time} \\
&= 7.5 \times \text{IC} \times \text{Clock cycle time} \\
\text{Speedup} &= 7.5
\end{aligned}$$

Assume that the CPI of a computer is 1 when all memory accesses hit in the cache. This is effectively an ideal scenario where you are not waiting for any memory request to be serviced. So, effectively memory stalls are 0, but if 30% of the instructions are loads and stores and the miss penalty is hundred cycles and the miss rate is 5%.

How much faster the computer be, if all instructions were cache hits? So, first start with the ideal scenario, when all the memory accesses are hit in the cache. So, our memory stall cycles equal to 0. So,

$$CPU\ time = CPU\ clock\ cycles * The\ clock\ cycle\ time$$

which is equal to,

$$CPU\ time = IC * CPI * Clock\ cycle\ time$$

where CPI is equal to 1. So, it is effectively IC into clock cycle time. Remember we have not given any clock frequency for the processor and we have also not given the number of instructions in the program.

So, effectively we consider IC and clock cycle time as it is. Now, consider the scenario where we have imperfect cache which has some percentage of memory request will incur miss and there is a miss penalty. So, memory stall cycles due to this, the miss rate is,

$$Memory\ Stall\ Cycles = IC * \frac{Memory\ access}{Instructions} * Miss\ rate * Miss\ penalty$$

And so overall CPU time because of this memory stalls is,

$$CPU\ Time_{Memory\ stalls} = \left(CPU\ Time_{Ideal\ Scenario} + CPU\ Time_{Memory\ stalls}\right) * IC * Clock\ cycle\ time$$

$$CPU\ Time_{Memory\ stalls} = 7.5 * IC * Clock\ cycle\ time$$

$$Speedup = 7.5$$

So, the speedup we achieve because of our perfect cache compared to an imperfect cache is equal to 7.5. So, this gives a motivation that we need to keep the useful data in the cache memory so that, the overall performance can be improved. And if you do not have a cache in this particular scenario for every request we have to go to the memory and which is going to take hundred cycle latency.

And so as a result the performance penalty will be significant, if we do not consider a cache memory, but whereas, if we consider a cache memory with some miss rate the performance can be improved compared to the case where no cache memory is considered. But of we have a cache memory which is perfect we can improve the performance even with respect to a cache memory with some percentage of misses and so on.

So, this motivates us to come up with the efficient cache memory between your processor and the main memory or we may have to have multiple levels of cache memory to improve the overall performance of the system. So, with this I am concluding this module and in the next module we are going to discuss the internals of the cache memory design.

Thank you.