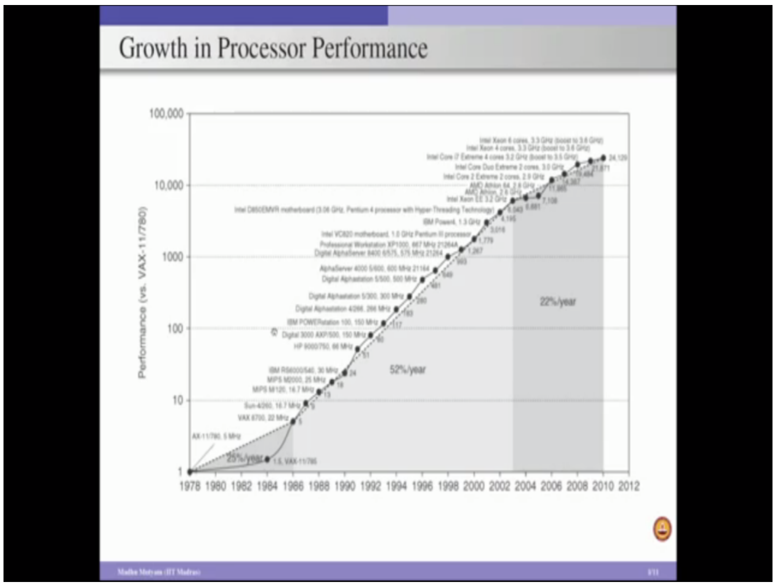
Computer Architecture
Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras.

Module – 02
Lecture - 02
Quantitative Principles of Computer Design

In the last module, we discussed the layered view of computer design as well as the overview of computer architecture course. So, in this module we look at the quantitative principles of computer design. So, before discussing the principles of computer design let us spend few minutes on looking at the growth in the processor performance, in the last 4 decades.

(Refer Slide Time: 00:39)



Growth in Processor Performance

So, this graph shows the growth in processor performance starting from 1978, all the way up to 2010. And the x axis shows the time line and the y axis is the performance with respect to VAX 11/780 machine. If you see the graph, the processor performance is increased almost like 25000 times. So, the main reason for this high performance improvement, in processors is because of two reasons - one is advancements in IC technology, the second one is advancements in computer design which is mainly contributed by architectural innovations.

So, if you see the graph you can find three regions - one is from 1978 to 1986, in this period the processor performance was almost 25 percent per year improvement. The second phase is 86 to 2003, where the processor performance increased significantly at a rate of 52 percent per year and in the last phase the performance was not so much improved as that of the previous periods, but there is some improvement like 22 percent per year.

And coming to architectural innovations in this first phase of the processor performance, it is mainly because of increase in the processor word size so started with 4 bit processors to 8 bit processor and so on. So, once you increase the processor word size you can perform operations much faster. In the second phase, the processor improvement is mainly because of architectural innovations, which are typically exploiting instruction level parallelism, which is in other words like executing instructions, independent instructions, in an out-of-order fashion, also implemented in Superscalar processors and also exploiting threading, it is a multithreading, hyper-threading, simultaneous multithreading and so on. Because of these things, coupled with multi levels of cache hierarchy, we were getting a significant improvement in the processor performance.

But in 2003, because of so many road blocks such as the power wall, frequency wall and the thermal, because of all this reasons, the processor performance was not significantly improved. Note that if you see the second phase of this graph, the increased clock frequency starting from 22 mega hertz all the way up to 3.2GHz. So, this is affecting like a thousand fold increase in the clock frequency, which actually contributed to a significant processor performance, but with the same trend we were not able to continue beyond 2003 because of the thermal issues.

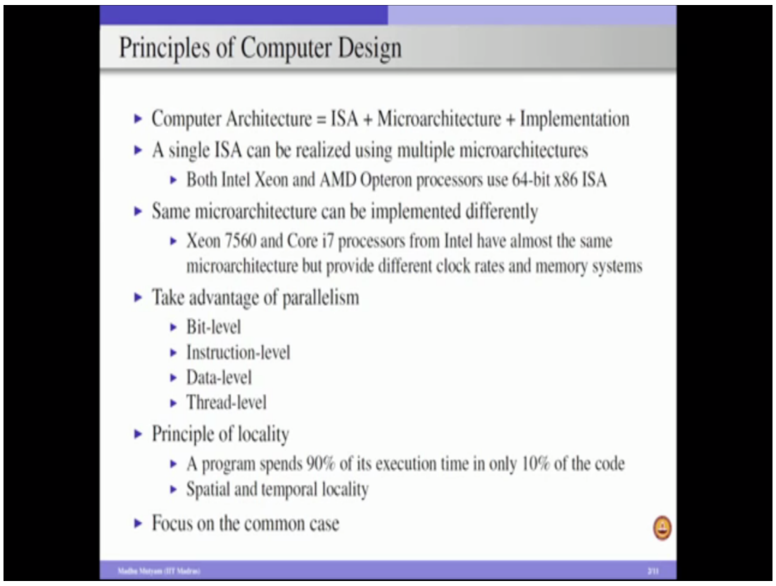
Remember that, because of the advancements in the fabrication technology, and the processor technology, we were able to integrate more and more transistors on a single chip. According to Moore's law, the number of transistors integrated in a chip doubles every 18 to 24 months. And he gave this idea almost like 50 years back and recently we have celebrated 50 years of Moore's law, even today the rule is holds true.

Now imagine a scenario, where you have billions of transistors on a single chip and all these transistors are working simultaneously and if you apply if you clock these transistors at 3.2GHz or more then they are going to consume too much power. And power per unit area is also called as power density, it increases significantly. And because of this power density, power density translates to heat dissipation and we will get the thermal issues.

And because of all these problems the technologists have decided not to increase the clock frequency at a rapid rate, as what they were doing the last twenty years or so. They said like rather than going for increasing the clock frequency, let us go for reducing the clock frequency or keeping the clock frequency at the same, at the same point, but have more collection of processors, each is a simple core and working together to compute a given problem efficiently. That is where the multicore era started. So, once you have a multicore processor, rather than a one complex processor to do a given task, now we can have two or four small processors working together and if the application is parallelizable efficiently, then we can improve the performance.

There is a main design paradigm change in 2003 and using this multi-core scenario, we were getting a performance improvement of almost 20 percent per year, after 2003. Now of course we are currently in multicore era, even if you see a mobile cell phone, we have like 2 core, 4 core and 8 core processors. Now we are in the multi-core era where the mantra is have collection of simple cores working together to solve a bigger problem efficiently. So, effectively, now if you can see the summary of this graph, it shows that the innovations in computer architecture play a key role in improving the overall processor performance or the system performance. Having said that, now we look at the principles of computer design. In the last module, we discussed that computer architecture is nothing but Instructions architecture, micro architecture and efficient implementation.

(Refer Slide Time: 07:43)



Principles of Computer Design

- Computer Architecture = ISA + Microarchitecture + Implementation
- A single ISA can be realized using multiple microarchitectures
  - Both Intel Xeon and AMD Opteron processors use 64-bit x86 ISA
- Same microarchitecture can be implemented differently
  - Xeon 7560 and Core i7 processors from Intel have almost the same microarchitecture but provide different clock rates and memory systems
- Take advantage of parallelism
  - Bit-level
  - Instruction-level
  - Data-level
  - Thread-level
- Principle of locality
  - A program spends 90% of its execution time in only 10% of the code
  - Spatial and temporal locality
- Focus on the common case

So, here we need to understand a point, given an ISA, we can come up with multiple microarchitectures. For example, Intel Xeon processor and AMD Opteron processor, uses the same ISA, 64-bit X86 ISA. So, that says that, based on the application requirements we come up with the functional requirements and we fix our ISA. But after fixing the ISA, we need to come up with an efficient microarchitecture because there are multiple microarchitectures you can realize for a given ISA.

Now, the second point is, once microarchitecture is fixed, it can be implemented differently. An example is a Xeon 7560 and core i7 processors from Intel, both have almost the same microarchitecture, but their implementations are different. When I say implementation is different, so they may clock at different frequencies and they may have different memory systems and so on. So, from these two points, it says that the computer architecture is nothing but the design space exploration because each microarchitecture and each implementation will have a different power performance points. So, you have a spectrum of design points and you need to select a good design point, based on your application requirements.

So, now if we come up with an efficient computer architecture, which in turn provides us an efficient computer. So, how do we come up with efficient computer architecture? What are the principles we need to follow? The first point is taking the advantage of parallelism. If you see the computer, or if you see the computer design, we can exploit the parallelism at various granularities starting from a finer granular point, which is a bit level parallelization, all the way up to very coarse grain parallelism which is the thread level parallelism.

If you see, if you recall the graph whatever we have shown in the previous foil, the architectural innovations constitutes all these things. In the initial phase of the processor performance graph, we saw that it is mainly because of bit level parallelization from the architectural point of view, we were getting an improvement. And when we come to the extreme end, the architecture innovations are mainly because of the core level parallelization, or the thread level parallelization. So, when I say a bit level parallelization rather than performing operations on let us say 4 bits, if we perform operations parallel on 8 bits, we can get more performance improvement.

So, effectively now, what is going to happen is, when I consider an 8 bit processor, if I want to perform two 64 and 16 bit number addition, I need to read two instructions one for LSB 8 bits and the other for MSB 8 bits. On the other hand, if I consider a 16 bit processor, I can do that in using a one instruction. So, this second point is instruction level parallelism, because most of the applications, if you see the instructions can be independent in some scenarios or even if they are dependent, we can overlap their operations, execution of the instruction can be overlapped. For example, if you see the lifetime of an execution of an instruction, the instruction will follow through a different stages. First is the Fetch, second is Decode, third one is Execute, Memory Read, Memory Write and finally the Commit ((Refer Time: 12:17)).

So, if you see here, when an instruction is in the Decode stage, the next instruction can be in the Fetch stage. Similarly, when the first instruction goes to data read, the second instruction will go to decode and third instruction will go to fetch stage and so on. So, effectively we can overlap the execution of instructions, this is called as a pipelining concept. We can exploit pipelining concept to improve the processor performance and this is especially true, when the instructions in some sense are dependent and so on. But, when the instructions are completely independent and so on, and if the processor supports multiple functional units and so on, we can actually execute these instructions independently and that is what the Superscalar processors are going to do.

We are going to discuss these pipelining and superscalar concepts, when we come to the corresponding modules. But the instruction level parallelism is one of the main contributors for a significant growth of processor performance. After this instruction level parallelism, the next level of parallelism we can exploit is at the data level. Example - if you consider a graphics application, we may not always need operations to be performed on a 32 bit data, and most of the times it may be on byte level data, 8 bit data and so on.

You can perform operations, a single operation on multiple data items. In other words operations can be performed on a data string, this is an example of SIMD - single instruction multiple data stream. Whatever we discussed in the last module the Flynn's classification of computer architecture, SIMD is one of the classifications. So, SIMD architectures typically exploit the data level parallelization and SISD exploits the instruction level parallelism.

And after this data level parallelization, the next component is the Thread level parallelization. Here if an application can be divided into multiple tasks or threads and if we have a hardware support to run these threads simultaneously, we can improve the performance at a rapid pace. So, all these independent threads can be executed on multiple cores and they will give you a significant improvement by reducing the overall computation time of that application. So effectively now, in order to exploit these different types of parallelization, starting from bit level all the way to the thread level, we need to have a support from the microarchitecture, the hardware, the ISA. In other words, the architecture needs to be supporting all these things to exploit these parallelizations. The next component is principle of locality.

What is principle of locality? Typically there is a thumb rule which says that a program spends 90 percent of its execution time in only 10 percent of the code. This says that, the data and instructions that we used recently will be required mostly in the near future. So, there are two types of principles of locality, one is the spatial locality and the other one is temporal locality.

What is spatial locality? Spatial locality indicates that, if we access some data or instructions now, there is a high chance that I will access the nearby data and instructions to that particular recently accessed element. Now temporal locality indicates that if I access an item now, there is a high chance that the same item will be accessed in the near future. So, how can we exploit this principle of locality?

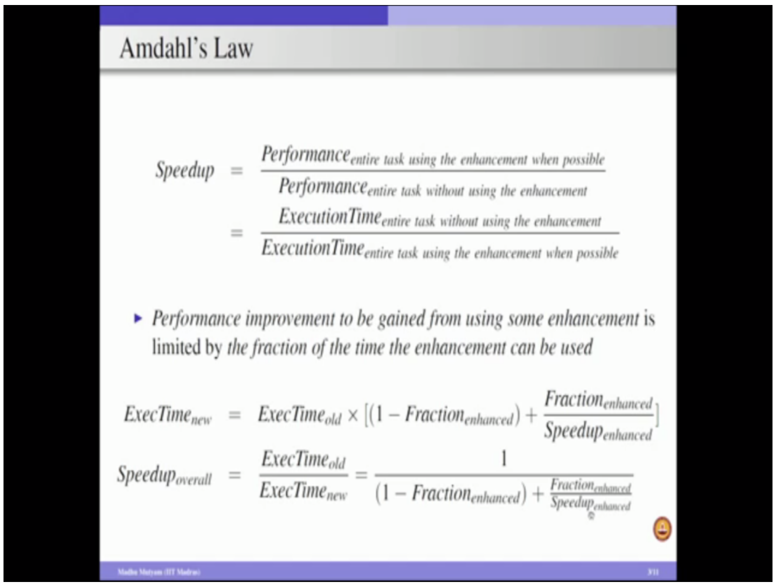
For that our microarchitecture needs to have a cache memory. We know that the data and instructions typically stored in the memory. Now always going to the memory to access these instructions and data will be very costly, because the processor frequency is much higher compared to the clock rate with which the memory is operating. So, as a result if you are always going to the memory, we are going to degrade the performance significantly.

So, as result what we can do is whenever we go to the memory, we bring a large chunk of data from the memory and put that in a fast memory, which is called as a cache memory, designed using SRAM ((Refer Time: 178:33)) based technology. When I say large chunk of data, it will not be in kilobytes or whatever, but typically we consider 32 bytes or 64 bytes of data. Anyway we are going to discuss the cache memory in detail in the next unit.

So, effectively we need to have the architecture that should exploit parallelism available at varied granularities. And also we need to have computer architecture, which need to support the cache memory. The third and the most important component for improving the performance or to come up with an efficient computer design is focusing on the common case. For example, consider a scenario where you have a pipelined processor where different stages of pipelines are - Fetch, Decode, Memory Read, Execute and Commit. And you also have set of functional units - adders, integers-multipliers, floating point multipliers and so on.

But if we see a scenario for executing most of the applications, we frequently access or use our fetch unit, decode unit, execute unit and so on, as compared to using the floating point multiplier. Given this scenario, we always have to put our efforts in optimizing this fetch unit, decode unit, execute unit and so on. Rather than putting our efforts on optimizing, a floating point multiplier. How do we quantify that? If I optimize a particular component, how much performance I can get? Because, we need to have some quantification mechanism so that before actually designing the processor, we will validate different alternatives and then come up with the best one. So, to do that we consider the famous Amdahl's law.

(Refer Slide Time: 19:51)



Amdahl's Law

$$Speedup = \frac{Performance_{entire\ task\ using\ the\ enhancement\ when\ possible}}{Performance_{entire\ task\ without\ using\ the\ enhancement}} = \frac{ExecutionTime_{entire\ task\ without\ using\ the\ enhancement}}{ExecutionTime_{entire\ task\ using\ the\ enhancement\ when\ possible}}$$

- *Performance improvement to be gained from using some enhancement* is limited by *the fraction of the time the enhancement can be used*

$$ExecTime_{new} = ExecTime_{old} \times \left[(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}\right]$$

$$Speedup_{overall} = \frac{ExecTime_{old}}{ExecTime_{new}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

Madhu Mutyam (IIT Madras) 3/11

Before describing what is Amdahl's law, let us define "Speedup".

$$Speedup = \frac{The\ performance\ of\ an\ entire\ task\ using\ an\ enhancement\ applied\ when\ possible}{The\ performance\ of\ the\ entire\ task,\ without\ using\ an\ enhancement}$$

Let us think we have an enhancement that we want to apply on a piece of code. When I apply, what is the performance I am going to get and without applying that what is the performance I am going to get and when I take the ratio of this then what I am going to get "Speedup". That shows that, we will get this much Speedup when I apply this particular enhancement on the piece of code.

Now, having defined the speedup, let us look at the Amdahl's law. Amdahl's law states that performance improvement to be gained from using some enhancement is limited by the fraction of the time the enhancement can be used. So, remember it has two components one is

"The performance improvement to be gained by using some enhancement" and the second one is "The fraction of time the enhancement can be applied". So, if the fraction of time the enhancement applied if 0, then you are not going to gain any performance improvement.

On the other hand, for the complete execution of the time if the enhancement is applied then you will get a significant improvement in the performance. So, by the way the performance is inversely proportional to the execution time. So, as a result the speedup can also be written as

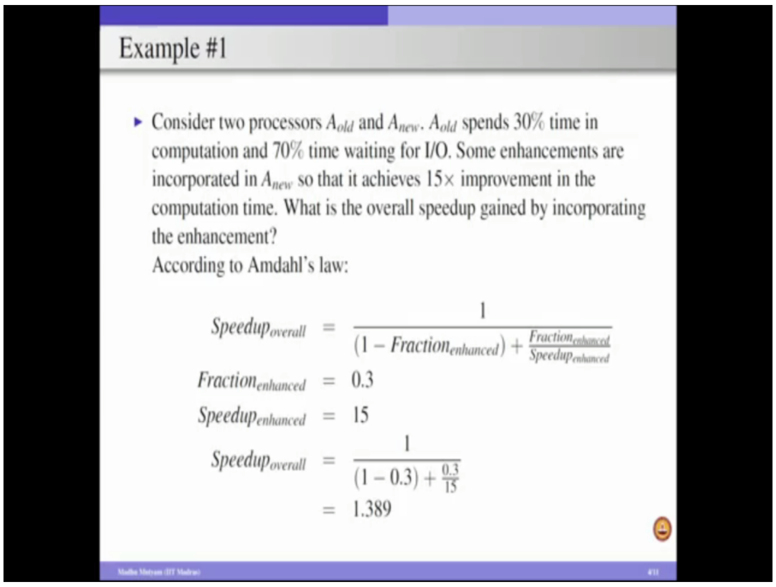
$$Speedup = \frac{Execution\ time\ for\ the\ entire\ task,\ without\ using\ any\ enhancement}{The\ execution\ time\ for\ the\ entire\ task,\ when\ using\ the\ enhancement\ when\ possible}$$

So, this Amdahl's law can be expressed as "the speedup achieved because of the enhancement applied" and "the fraction of time the enhancement is possible" which is nothing but it is -

$$Speedup = \frac{1}{\left(1 - Fraction\ of\ the\ time_{Enhancement\ is\ applied}\right) + \frac{Fraction\ of\ time_{Enhancement\ applied}}{The\ speedup\ achieved\ because\ of\ enhancement}}$$

We know that speedup enhanced is calculated by this particular formula, so this shows that even when you apply enhancement, your overall performance is actually dictated by the fraction of the time the enhancement is considered.

(Refer Slide Time: 22:51)



Example #1

- Consider two processors A_{old} and A_{new}. A_{old} spends 30% time in computation and 70% time waiting for I/O. Some enhancements are incorporated in A_{new} so that it achieves 15× improvement in the computation time. What is the overall speedup gained by incorporating the enhancement?
  According to Amdahl's law:

$$\begin{aligned} Speedup_{overall} &= \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}} \\ Fraction_{enhanced} &= 0.3 \\ Speedup_{enhanced} &= 15 \\ Speedup_{overall} &= \frac{1}{(1-0.3) + \frac{0.3}{15}} \\ &= 1.389 \end{aligned}$$

Madhu Mutyam (IIT Madras) 4/11

So, let us consider few examples to understand this Amdahl's law and then comparing design alternatives for different processors. The first example is to consider two processors – A_{old} and A_{new}, and A_{old} spends 30% of the time in computation and 70% of the time waiting for I/O operations. Some enhancements are incorporated in A_{new} Processor, so that it achieves 15 times improvement in the computation time. What is the overall speedup gained by incorporating the enhancement to know whether A_{new} is better or A_{old} is better. So, we need to find the performance achieved with A_{new} and $A_{old,}$ for that we use the Amdahl's law whatever we discussed in the previous slide.

According to Amdahl's law,

$$Speedup\,Overall = \frac{1}{\left(1 - fraction\,of\,enhanced\right) + \frac{fraction\,enhanced}{speedup\,enhanced}}$$

This is effectively like you can say,

$$Speedup\,Overall = \frac{1}{Fraction\,of\,component_{enhancement\,not\,applied} + \frac{Fraction\,of\,time_{enhancement\,is considered}}{Speedup\,achieved\,because\,of\,the\,enhancement}}$$

And from the data given in the problem statement,

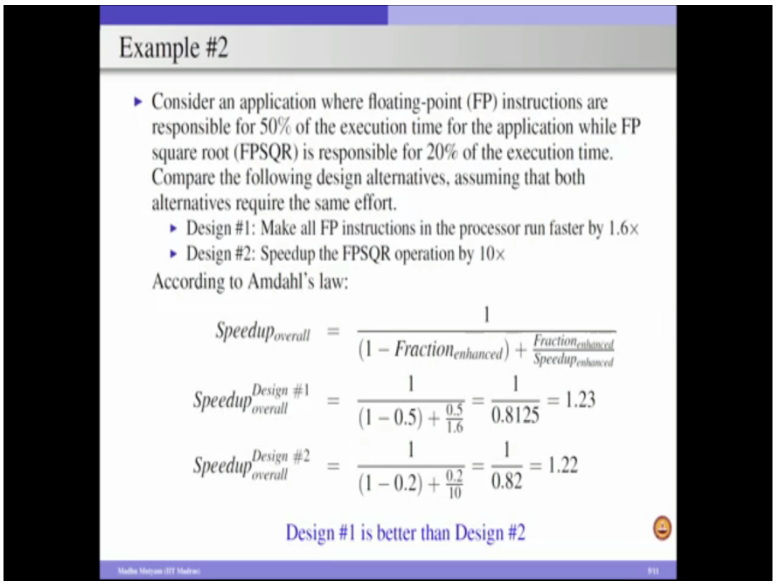
$$Fraction\,Of\,Enhanced = 0.3$$

because 30% of the computation on 70% time is spent for the I/O. And we are applying the enhancement only for the computation part. So, effectively our enhancement applied to 0.3% of your total execution time and 0.7% of the time, where no enhancements are applied and when I apply the enhancement, we know that the speedup we achieve is 15 times. Substituting these values, so we will get -

$$Speedup\,Overall = 1.38$$

This is effectively says that A_{new} processor will be having 1.389 times speedup with respect to A_{old}. So, effectively we can go for this enhancement and improve the performance.

(Refer Slide Time: 25:09)



Example #2

- Consider an application where floating-point (FP) instructions are responsible for 50% of the execution time for the application while FP square root (FPSQR) is responsible for 20% of the execution time. Compare the following design alternatives, assuming that both alternatives require the same effort.
  - Design #1: Make all FP instructions in the processor run faster by 1.6×
  - Design #2: Speedup the FPSQR operation by 10×

According to Amdahl's law:

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

$$Speedup_{overall}^{Design\ \#1} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

$$Speedup_{overall}^{Design\ \#2} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

Design #1 is better than Design #2

Madhu Mutyam (IIT Madras) 9/11

Consider second example, application where floating point instructions are responsible for 50 percent of the execution time for an application while floating point square root is responsible for 20 percent of the execution time, and compare the following design alternatives, assuming that both alternatives require the same effort. Design 1 is make all floating point operations in the processor run faster by 1.6 times, the design 2 is speedup floating point square root operation by 10 times speedup.

So, if we just without computing, if we just look at this design 1, design 2 and we may be tempted to say that floating point square root operation will be better, because it gives a 10 times speedup versus1.6 speedup for the design 1. Now let us see whether that is true or false. According to again Amdahl's law,

$$Speedup_{Overall} = \frac{1}{Fraction\ of\ time_{enhancement\ not\ applied} + \frac{Fraction\ of\ time\ where\ enhancement\ applied}{Speedup\ achieved\ with\ the\ enhancement}}$$

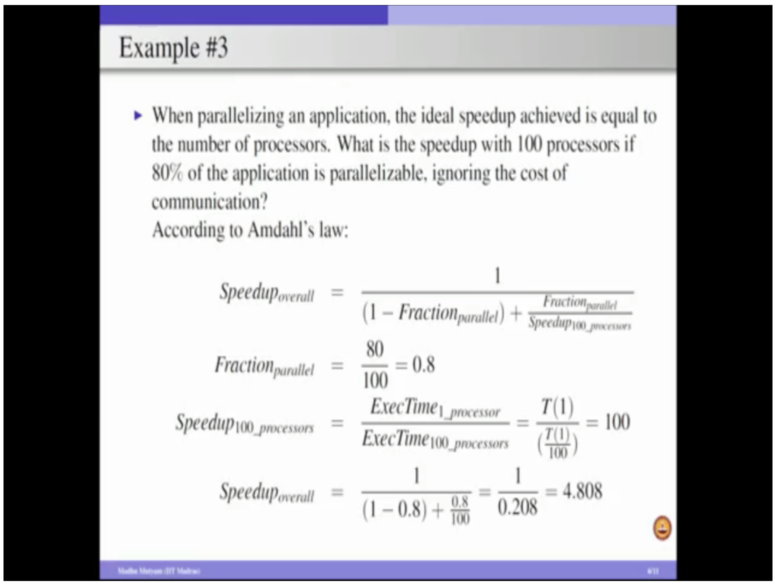
For the design 1, where we make all floating point operations, execute much faster by 1.6 times and we know that floating point operations constitute 50% of the execution time.

So effectively,

$$Speedup_{Overall} = \frac{1}{1-0.5+\frac{0.5}{1.6}} = 1.23$$

Overall speedup we achieve with the design 1 is 1.23 with respect to the base processor. And design 2 we know that floating point square root operations constitutes 20% of the execution time. So, effectively 0.8% of the time no enhancement applied and 0.2 times we get an improvement of 10x in executing the floating point square root operations. So, when we substitute these values in the Amdahl's law, we get the speedup of 1.22. So, this shows that design 1 is better than design 2. Remember in these two examples, we consider the optimizations incorporated within a processor, but actually Amdahl's law can be applied across processors also.

(Refer Slide Time: 27:43)



Example #3

- When parallelizing an application, the ideal speedup achieved is equal to the number of processors. What is the speedup with 100 processors if 80% of the application is parallelizable, ignoring the cost of communication?
  According to Amdahl's law:

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{parallel}) + \frac{Fraction_{parallel}}{Speedup_{100\_processors}}}$$

$$Fraction_{parallel} = \frac{80}{100} = 0.8$$

$$Speedup_{100\_processors} = \frac{ExecTime_{1\_processor}}{ExecTime_{100\_processors}} = \frac{T(1)}{(\frac{T(1)}{100})} = 100$$

$$Speedup_{overall} = \frac{1}{(1 - 0.8) + \frac{0.8}{100}} = \frac{1}{0.208} = 4.808$$

Madhu Mutyam (IIT Madras) 6/11

To illustrate that, let us consider a simple example when parallelizing an application, the ideal speedup achieved is equal to the number of processors. What is the speedup with 100 processors if 80 percent of the application is parallelizable, ignoring the cost of communication. Let us consider an ideal scenario where communication cost is 0 and in an ideal setup, if I have n processors and if my code is thoroughly parallelizable I can get a speedup of n.

Now, in this particular example we consider 100 processors. Now, we will see, what is the improvement we will get with these 100 processors when my code is 80% of parallelizable. So, we can restate Amdahl's law as speedup overall is equal to

$$Speedup_{Overall} = \frac{1}{1 - Faction\,of\,time\,parallelizable + \frac{Fraction\,of\,time\,parallelizable}{Speedup\,achieved\,with\,the\,parallelization}}$$

It is nothing but,

$$Speedup_{Overall} = \frac{1}{Sequential\,Portion\,of\,the\,Code + \frac{\text{Parallel}\,portion\,of\,the\,code}{Speedup\,achieved\,with\,number\,of\,processors}}$$

Note that when the code is sequential even when you have 1000 processors, you have to execute the sequence of instructions in the sequential portion, in a serial fashion. So, you would not get any speed, but whereas, if the portion of the code is parallelizable and when we assume that the communication cost is 0, and in an ideal scenario where every instructions executed takes 1 unit of time and so on. So, we can get, with 100 processor we can get, 100x speed.

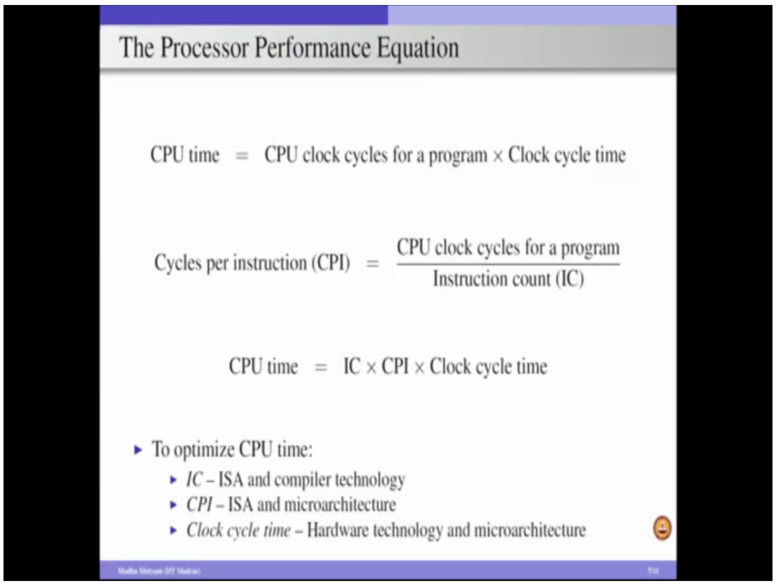
So, substituting these values in this equation, so fraction of parallel portion is 0.8, sequential portion is 0.2 and effectively the speedup with 100 processors is -

$$Speedup_{With\,100\,Processors} = \frac{T(1)}{\left(\frac{T(1)}{100}\right)}$$

where T1 is "The total time it takes for a single processor to complete the task". And the speedup is 100 here and the speedup overall is 4.8 which shows that even when you have 80% of the code parallelizable, even with 100 processors at your disposal you can get not more than 5 percent improvement in performance.

So, this also shows that, to have an efficient computer design, in addition to having efficient computer architecture, we need to have support from the different other layers of the computer design. Those are like algorithmic layer, high level programming layer, the operating system layer, the system software everything. We need to have an efficient compiler, we need to have an efficient program, and we need to write a parallel program. So, all these things actually contribute to the overall efficient design of a computer. We will use a processor performance equation.

(Refer Slide Time: 31:03)



The Processor Performance Equation

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

$$\text{Cycles per instruction (CPI)} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count (IC)}}$$

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{Clock cycle time}$$

- To optimize CPU time:
  - *IC* – ISA and compiler technology
  - *CPI* – ISA and microarchitecture
  - *Clock cycle time* – Hardware technology and microarchitecture

We define a CPU time as nothing but “The total number of CPU clock cycles an application takes when we are executing that application on a given processor”. Remember, if I just give you CPU clock cycles without considering the cycle time, it makes no sense because, I may say 100 CPU cycles, on one processor for such 1000 CPU cycles on another processor. And if the first processor is running at 10 MHz frequency and the second one is working with 1 GHz frequency. So, which one is better? Actually the second one is better, even though the number of clock cycles it takes is more. The reason is the second processor is working at 1GHz frequency whereas the first one is working at 10 MHz.

So, in other words when we specify the clock cycles, we need to give the cycle time also. Effectively, the total CPU time for an application to execute on a given processor is equal to the product of the number of CPU cycles and the CPU cycle time or the clock cycle time. So, once we consider the CPU time, we also can measure the number of instructions in a given program that is denoted as instruction count or IC. Once we have an instruction count and also the total number of CPU clock cycles for a program, we can define clock cycles per instruction or CPI. CPI is the ratio of “CPU clock cycles for a program” and “instruction count”.

So, when we substitute this in the first equation,

$$CPU\ time = Instructio\ count * Clock\ cycles\ per\ instruction * Cycle\ time$$

So, effectively given a program, and also given the characteristics of a processor, we can identify what is the clock cycle time, we can identify what are the number of instructions in the program based on our ISA, and we can calculate the CPI when we execute program on the processor. So, to optimize the CPU time because ultimately our overall idea is to improve the overall performance, to improve the performance you need to reduce the execution time.

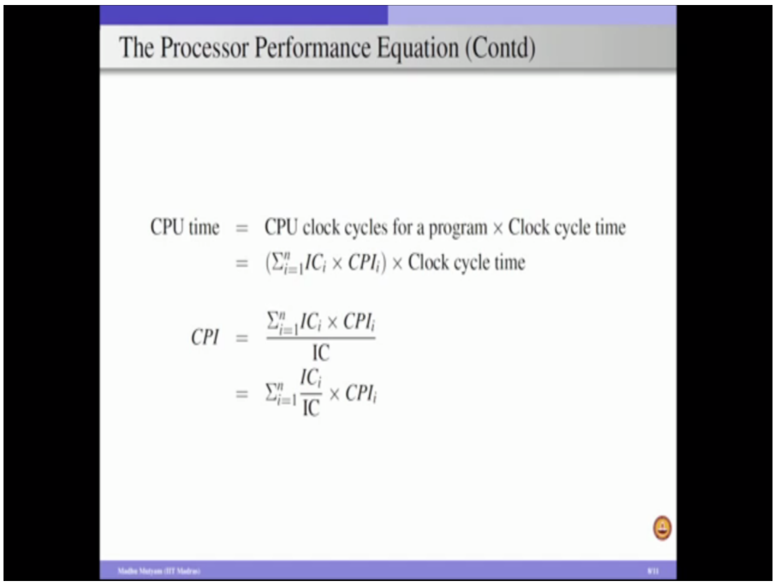
In other words, we need to reduce the CPU time. To reduce the CPU time, either we can optimize instruction count or we can optimize CPI, or we can optimize the clock cycle time. But this is not that straight forward. If you try to optimize one of these things, it has an impact on the other two components. The reason is IC - Instruction Count is actually dependent on the given ISA and the compiler technology. So, we can come up with a complex instruction to perform a complex task, versus a simple set of instructions which may require four instructions to perform the task.

Now, in the complex instruction scenario, my IC is equal to 1, in the simple instruction scenario my IC is equal to 4. So, if I go for a complex instruction then IC is equal to 1, but the problem with that is because the complex instruction is considered, it is going to increase your clock cycles per instruction. So, effectively when I am trying to reduce optimize one component, it has an impact on the other component. Clock cycle time depends on the hardware technology and the microarchitecture.

So, this effectively shows that each of these components are determined by multiple different components. So, we have to be intelligently optimizing these things without having a significant impact on the other components then only we can improve the overall performance of the system. So, we know that -

$$CPU\ time = IC * CPI * Clock\ Cycle\ Time$$

(Refer Slide Time: 35:50)



The Processor Performance Equation (Contd)

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$
$$= \left(\Sigma_{i=1}^{n} IC_i \times CPI_i\right) \times \text{Clock cycle time}$$

$$CPI = \frac{\Sigma_{i=1}^{n} IC_i \times CPI_i}{\text{IC}}$$
$$= \Sigma_{i=1}^{n} \frac{IC_i}{\text{IC}} \times CPI_i$$

Madhu Mutyam (IIT Madras)

This can also be written as -

$$CPU\ time = \left(\sum_{i=1}^{n} IC_i * CPI_i\right) * Clock\ cylce\ time$$

For example, I have an application, which consists of add instructions and multiply instructions, 80 percent of instructions are add instructions and 20 percent of instructions are multiply instructions.

Now if I want to compute the CPU time for that particular application all I have to do is,

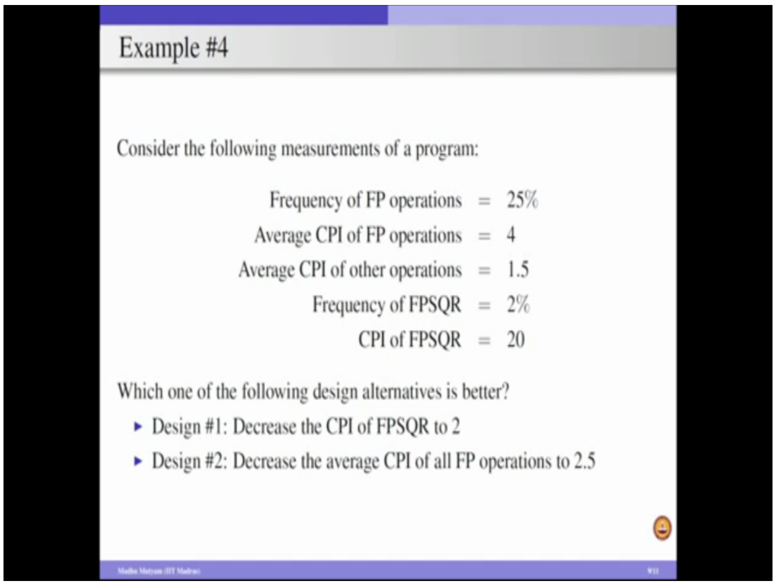
$$CPU\ time = 0.8 * CPI\ of\ add\ Instruction + 0.2 * CPI\ of\ Multiply\ Instructions$$

Take the sum and multiply with the cycle time. This can be written as -

$$IC_i / IC$$

where IC_i is instruction count of a particular instruction type by the total number of instructions in the program. So, effectively I take the fraction of the instruction type, multiplied with the corresponding CPI and take the sum of all in the program, and then multiply with cycle time, you will get the overall CPU time.

(Refer Slide Time: 37:06)



Example #4

Consider the following measurements of a program:

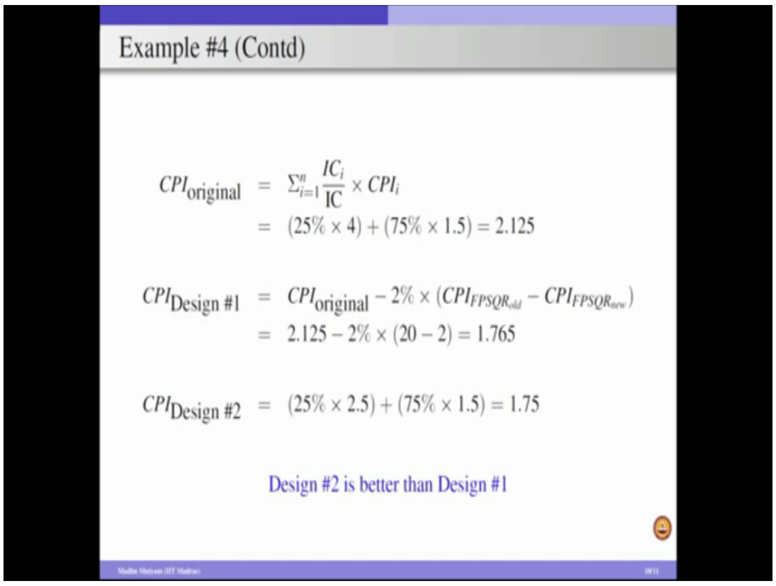
Frequency of FP operations	=	25%
Average CPI of FP operations	=	4
Average CPI of other operations	=	1.5
Frequency of FPSQR	=	2%
CPI of FPSQR	=	20

Which one of the following design alternatives is better?

- Design #1: Decrease the CPI of FPSQR to 2
- Design #2: Decrease the average CPI of all FP operations to 2.5

Let us consider an example, Consider an application where the frequency of floating point operations is 25 percent, and the average CPI for the floating point operations is 4. Averages CPI of all other operations is 1.5 and assume that the frequency of floating point square root operations is 2 percent, and the CPI of floating point square root operations is 20. Now, whether decreasing the CPI of floating square root operations to 2 is better as compared to decreasing the average CPI of all floating point operations to 2.5. So, how do we compare these two design alternatives and suggest the best one. To do that we use the processor performance equation.

(Refer Slide Time: 37:56)



Example #4 (Contd)

$$CPI_{\text{original}} = \sum_{i=1}^{n} \frac{IC_i}{\text{IC}} \times CPI_i$$
$$= (25\% \times 4) + (75\% \times 1.5) = 2.125$$

$$CPI_{\text{Design \#1}} = CPI_{\text{original}} - 2\% \times (CPI_{FPSQR_{old}} - CPI_{FPSQR_{new}})$$
$$= 2.125 - 2\% \times (20 - 2) = 1.765$$

$$CPI_{\text{Design \#2}} = (25\% \times 2.5) + (75\% \times 1.5) = 1.75$$

Design #2 is better than Design #1

We know that,

$$CPI_{Original} = \sum The\, instruction\, count\, fraction * corresponding\, CPI$$

So, in the original case, we have -

25% of the instructions are floating point which takes 4 cycles as CPI + rest of the instructions are 75% which takes 1.5 as CPI is effectively 2.125. And CPI of design 1, the design 1 which says that we can go for optimizing floating point square root operations, so that it reduces the CPI from 20 to 2.

So, the CPI of design 1 is equal to 1.765. Yes, this is better than the original CPI, but it is not saying anything about whether it is better with respect to design 2. Let us see, what is the CPI of design 2. So, a design 2 is, we optimize all floating point operations and the average CPI of all floating point operations now reduce to 2.5 as compared to the original 4, and they have not touched any of the other non-floating point operations, which have CPI of 1.5. So, it is the total CPI for the design 2 is 1.75, so which says that the design 2 is better than design 1.

Remember, when we say design 2 is better, because the CPI is reduced. Because lower the CPI, we are going to get the higher performance, because we spend minimum number of cycles to execute an instruction. So, lower is better, when we consider CPI as metric. So effectively, design 2 is better, in other words optimize all floating point operations rather than considering floating point square root operations. This also again comes back to our previous discussion; concentrate on the common case because floating point operations constitute 25 percent of the total operations whereas, the floating point square root contributes only 2 percent of the total instructions. So, with this I am concluding this module. Next module, we are going to discuss the instruction set architectures.

Thank you.