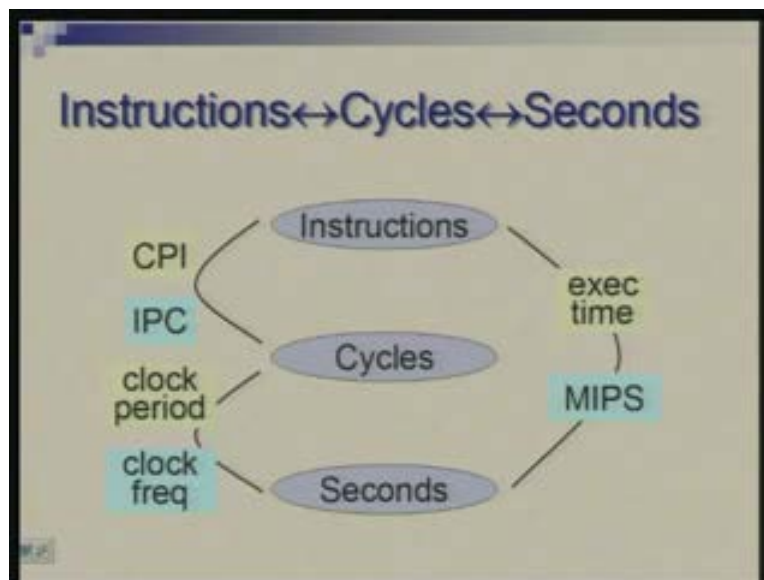


**Computer Architecture**  
**Prof. Anshul Kumar**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**  
**Lecture - 10**  
**Performance (contd..)**

We will continue with our discussion on performance evaluation of processors. In the last lecture we saw that there could be different angles or different perspectives when you look at the performance issue and therefore the definitions may change. We tried to follow the definition which would be true for individual user's perspective and from that point of view we try to define performance in a quantitative manner essentially trying to say that it is execution time of a given program over an architecture over a machine which matters. We will go further into this, look at more examples and try to see how this is done professionally so that a large community of architects and users could share same ideas.

(Refer Slide Time: 1:54)



So essentially the formula or the definition we came up for performance tried to interlink three aspects: the number of instructions which you require to execute a program, the number of clock cycles which are required to execute a program and the total time which is spent in executing the program. So these are the three ways or three aspects which are of concern to you. The one at the bottom is the number of seconds; how much time, how many seconds or minutes or hours are taken to run a program. This could also be expressed in terms of number of clock cycles because everything which is done in a computer is in terms of clocks. And also, one could say in a slightly higher level that everything is done in terms of instructions.

So the time taken could be measured in these three units where this is more abstract (Refer Slide Time: 2:56), less abstract and least abstract. So this is a real measure and there are quantities which we define which link these.

So, for example, the relationship between instructions and cycles could be expressed as cycles per instructions; number of cycles you require to do an instruction, you can talk of individual instructions, you can talk of collection of instructions in terms of its average. So when you are talking of entire program what would matter is average CPI or average cycles per instruction for that particular program.

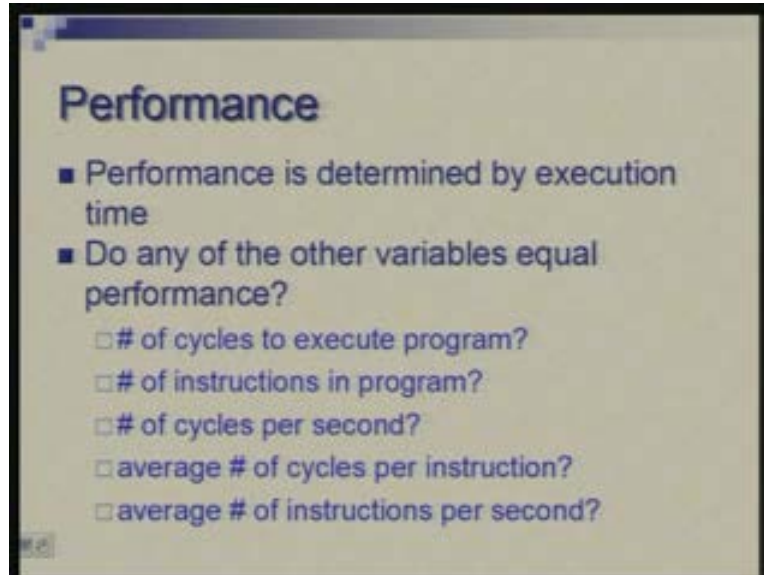
Now remember that this average could be different from program to program. The same thing actually said in an inverse manner is often called IPC or instructions per cycle. So the notion here is different that as a few are trying to do multiple instructions in a cycle. There are machines, there are processors which actually are at that level of performance where they do multiple instructions per cycle and there it becomes perhaps more important to talk of instructions per cycle. But for this particular course where we are talking of basic architectures we will essentially confine ourselves to CPI. One could say that the two are reciprocal of each other but it is the perspective which is different.

Then what relates cycles to the absolute time is the clock period or conversely clock frequency so that is something very straightforward. Finally the relationship between instructions and the absolute time; you could talk of the total time. Each instruction takes seconds per instruction or you can also talk of instructions per second. So when you are talking of time seconds per instruction there is another thing which we actually talked last time which was seconds per program.

For the entire program how many seconds are required or for one instruction how many seconds are required one should ask both questions. So when you are talking of time per instruction it is actually talking of instruction rate that may not really capture the entire picture of how long a program takes a program may execute at a fast rate but the total instruction may be large so it is the overall time which matters. But still often one talks about the rate at which instructions are done that is instructions per second and says this to make a more convenient unit you talk of million instructions per unit or MIPS. This is the average rate at which instructions get executed (Refer Slide Time: 5:41). Reciprocal of this would be the number of micro seconds you are spending per instruction.

So one could talk of MIPS as an average. People have also talked of peak MIPS **which is because the** if you look at the instantaneous rate at which instructions are done it could vary; there are slow instructions, there are fast instructions so peak MIPS would refer to.... if you were doing only the fastest instructions what is the rate at which you will do it.

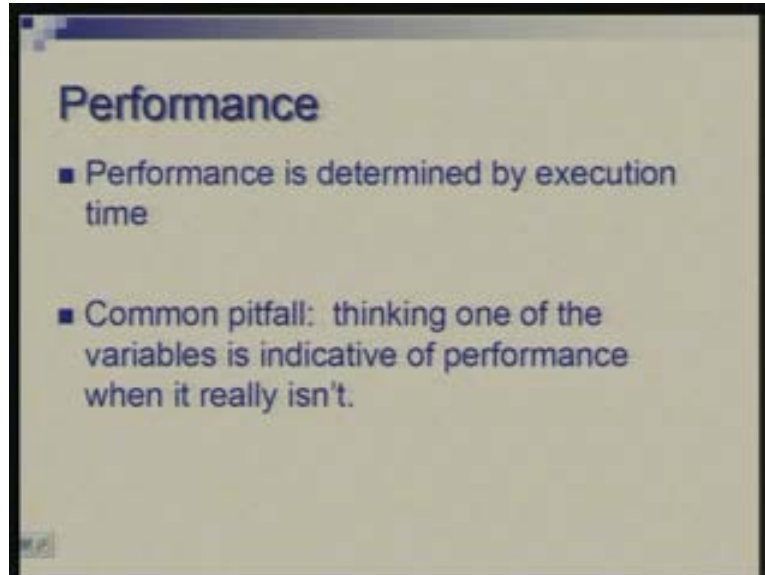
(Refer Slide Time: 6:30)



Thus, in nutshell what we have said is that performance is determined by execution time particularly an individual programmer or individual user's perspective. At different stages people have talked of other ways of looking at it and the question is do any of the other variables truly represent performance and some of these possibilities are: number of cycles to execute the program, the number of instructions in the program, number of cycles per second, average number of cycles per instruction, average number of instructions per second. So I have talked of many of these things but ultimately it is this which is important the execution time.

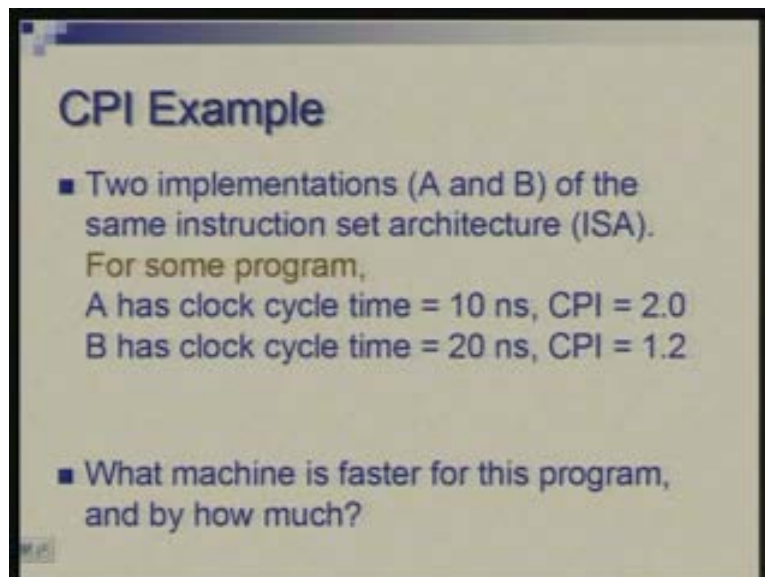
Here when I am talking of number of instructions in program this is a factor which is often looked upon in two different ways: one could be the length of the program. As you see statistically a program has so many instructions as how many mega bytes or kilo bytes it occupies so that is linked to the number of instructions it has. But when you are talking of performance you are talking of dynamic number of instructions; that number of instructions executed would be the term. So, if the program has some loops of course some instructions gets repeatedly so from performance point of view you have to count those those many times or if they are conditional like something gets skipped or something gets executed you have to see what gets executed. So, to do some computation for a given data program may execute certain number of instructions and that is what one is often talking of.

(Refer Slide Time: 8:00)



So a common pitfall is thinking of other possible indicators which are not true indicators of performance; they may be related but they are not true indicators. Now let us look at a few examples. We looked at last example one example in the last lecture towards the end; we will see a few more today.

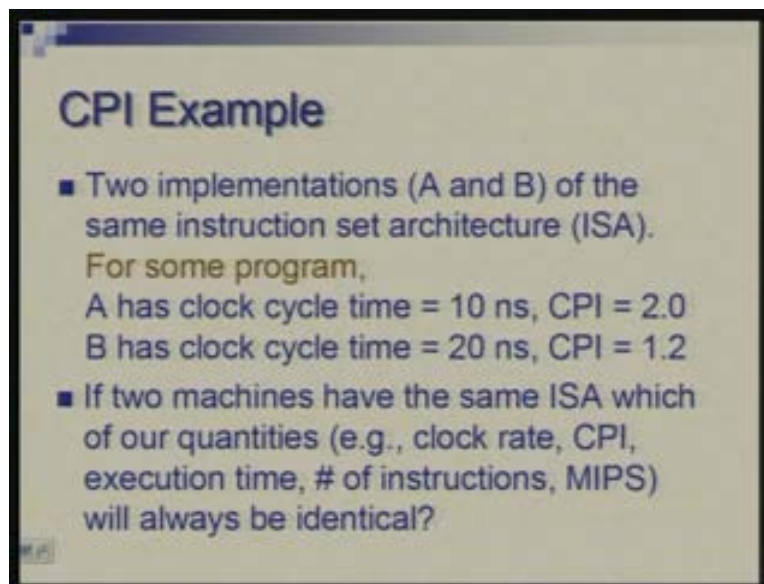
(Refer Slide Time: 9:25)



Suppose we are talking of two implementations of the same instruction set architecture or ISA let us call these A and B. Instructions are same but there are two different ways of implementing them in hardware. Now, for a given program suppose A implementation has a clock cycle time of 10 nanoseconds and B has 20 nanoseconds the CPI the rate at

which instructions are executed or number of cycles per instruction on average is 2.0 in case of A and 1.0 1.2 in case of B. So the question is that for this particular program which machine or which implementation is faster. So we can apply the formula which says that you take the number of instructions executed multiplied by CPI and multiply it by clock cycle time. So one question you need to answer is that if these two machines as the case now which have same ISA which of the quantities remain equal..... number of instruction because it is the same program expressed in the same instruction set which we are talking of so it is the number of instructions which will remain same, other things could vary; clock rate is given to be varying, CPI is given to be varying, MIPS or the number the rate at which instructions are done will vary and so would be the execution time. It is only the instruction the number of instructions executed that will remain unchanged.

(Refer Slide Time: 9:47)



### CPI Example

- Two implementations (A and B) of the same instruction set architecture (ISA).  
For some program,  
A has clock cycle time = 10 ns, CPI = 2.0  
B has clock cycle time = 20 ns, CPI = 1.2
- If two machines have the same ISA which of our quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will always be identical?

Now we are not given the number of instructions in this case. All we know is the two are same so our objective is just to find relative performance or just find which is faster which is slower so that can be done and therefore we can knock off that common factor which is number of instructions. essentially you take product of these figures 10 nanoseconds and 2.0 which basically says it is 20 nanoseconds is the average time you are spending per instruction and to the same instructions where 24 is the 24 nanoseconds is the time you spent per instruction in the second case and clearly A is faster.

(Refer Slide Time: 12:22)

**# of Instructions Example**

A compiler designer is to decide between two code sequences.

- First code sequence:
  - 2 of A, 1 of B, 2 of C
- Second code sequence:
  - 4 of A, 1 of B, 1 of C

3 classes of instructions:

- A : 1 cycle
- B : 2 cycles
- C : 3 cycles

Another example: here situation would be such that the number of instructions would vary. So suppose put yourself in the shoes of compiler designer who has to decide between two alternative sequences of code to translate a given high level language computation. There is some statement in high level language it can be translated in two alternative ways and one has to make a choice between these two. So the architecture which is available to him the instruction set architecture are first three classes of instructions: A B and C and they differ in terms of course their functionality but in terms of cycles the number of cycles spent is 1, 2 and 3 in the three cases.

Class A instructions take 1 cycle; class B 2 cycles and class C 3 cycles each. So the options which the compiler designer is debating are these two code sequences. First sequence uses 2 instructions of type A, 1 instruction of type B and 2 instructions of type C for same computation so this is one possible way one could translate same computation. Second way is 4 of A, 1 of B and 1 of C.

So now how do you evaluate this choice?

You can find out the time which will take for both these cases and you can see that in first case that is the same sequence the sequence will require a total of 2 plus 2 plus 6 so that is 10 cycles and in the second case it will require 4 1 and 1 so which means the 4 plus 2 8 and 3 [students : 9] sorry what did I do; 4 plus 2 6 and 3 9 yeah and so second is faster although the total number of instructions in second one is longer. So, if you were to just compare the number of instructions first sequence is of 5 instructions second is of 6 if you do not go into timing you might say first sequence is shorter and one may be tempted to do that. But only when you look into the cycle information you realize that essentially what second sequence does is it tradesoff one slow instruction for two fast instructions and still makes sense.



(Refer Slide Time: 14:11)

**# of Instructions Example**

A compiler designer is to decide between two code sequences.

- First code sequence:
  - ☐ 2 of A, 1 of B, 2 of C
- Second code sequence:
  - ☐ 4 of A, 1 of B, 1 of C

3 classes of instructions:

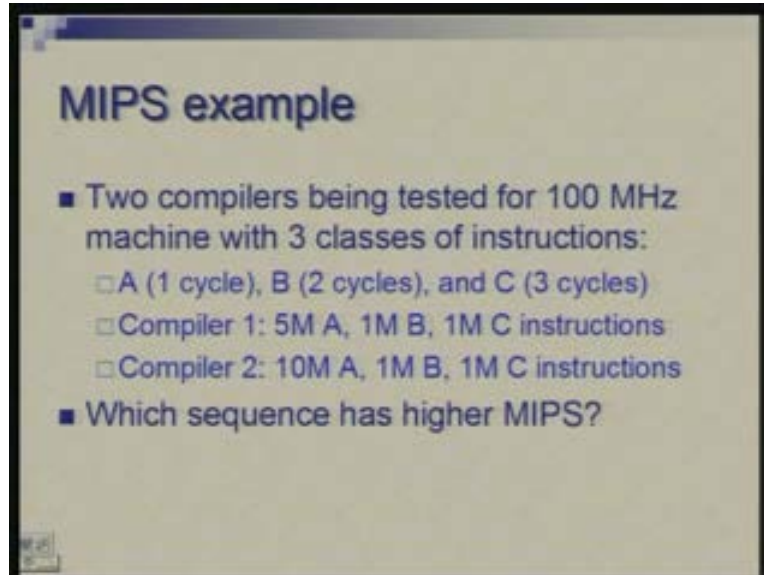
- ☐ A : 1 cycle
- ☐ B : 2 cycles
- ☐ C : 3 cycles

- Which sequence will be faster?
  - ☐ How much?
  - ☐ CPI for each?

Therefore, one could quantify the difference how much is sequence 2 faster so it is 9 versus 10 and you can also find CPI for each. So what is the **average** average CPI in the two cases can you identify it quickly? 2, it will be..... basically you add these you take average of these with these as weightage (Refer Slide Time: 14:42) so it is 2 plus 2 plus 6 so basically..... we have already counted the number of cycles we can divide by number of instructions. So 10 cycles of 5 instructions which means 2 so CPI on the average is 2 and here you have 9 cycles divided by 6 so you have 1.5.

Yet another example. So we have two compilers which are being tested for a machine. Again there is kind of similarity. Once again the compiler is the one which is varying. So the clock frequency is 100 MHz and there are three classes of instructions. It is a similar situation 1, 2 and 3, three classes are A B and C. So compiler one produces..... **for a** for a given program, now we are looking at the entire program; the situation is similar but instead of some piece of code we look at one whole program. So compiler 1 produces 5 million instructions of type A, 1 million of type B, 1 million of type C and second compiler produces 10 million of type A, 1 million of type B and 1 million of type C so it is just producing more instruction of type A and others are same.

(Refer Slide Time: 16:47)

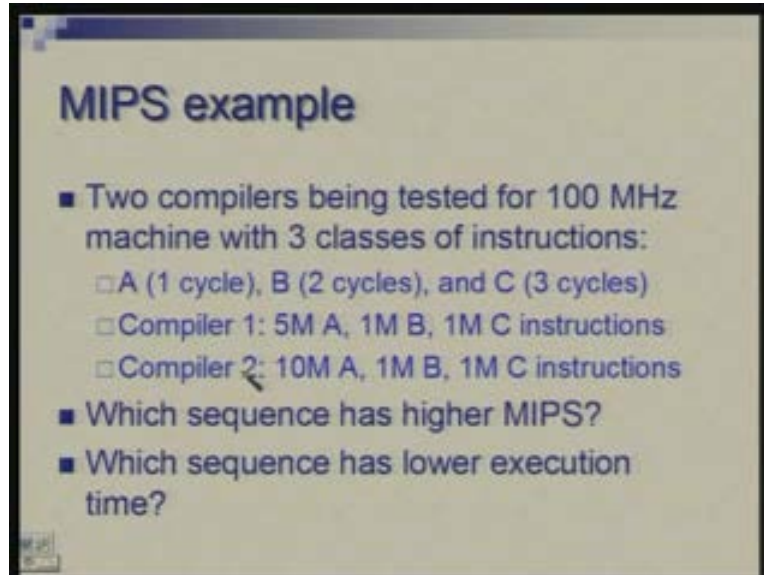


Now which compiler gives you higher MIPS? The first, let us say, **you can** you can add these; add the cycles of these instructions and you know that the cycle time or the rate at which cycles are running is 100 MHz so total of cycles required for these instructions will be 5 million plus 2 million that is 7 and 3 million that is another 3 so totally 10 million cycles are required and each cycle takes 0.1 nanoseconds.

[Conversation between student and professor.....(17:33) let me see, no, we are trying to talk of instructions per second]..... so 0.1 nano sorry 10 nanoseconds multiplied by 10 million cycles is how much..... let us see it is 100 so 1 by 1000 so 1 by 1 by 10 seconds is what you would require. **So you have..... am I,..... yeah.....** So you have executed 7 million instructions in 1 by 10 seconds **is that right** 1 by 10? Just check my calculations. So 7 million instructions in 1 by 10 seconds. So what is the instruction rate? It is a 70 MIPS so 70 million instructions per second.



(Refer Slide Time: 18:59)



### MIPS example

- Two compilers being tested for 100 MHz machine with 3 classes of instructions:
  - A (1 cycle), B (2 cycles), and C (3 cycles)
  - Compiler 1: 5M A, 1M B, 1M C instructions
  - Compiler 2: 10M A, 1M B, 1M C instructions
- Which sequence has higher MIPS?
- Which sequence has lower execution time?

Let us do it for the second case. So the number of cycles here is 10 million for this, 2 million for this and 3 million for this so basically 15 million and the total time spent would be 15 **sorry** it should be 1.5 times that so 0.15 0.15 seconds as opposed to 0.1 seconds. In case 1 we spent 0.1 seconds now we are spending 0.15 seconds and the number of instructions being done is 12 million. So what is the MIPS we are getting? 80 MIPS. So the answers are 70 MIPS in the first case and 80 MIPS in the second case. So, if you were to take MIPS as the comparison criteria it might appear that in the second case you take the second program generated by the second compiler it is giving you faster MIPS more MIPS but it is obvious that the total time spent here is more and therefore compiler 2 is producing a poor report. So execution time is lower in first case so in that sense its better whereas MIPS is higher in second case so apparently the second may look better but one has to be careful here. **So is it clear?**

(Refer Slide Time: 21:03)

**Performance Example**

- M1 and M2 are two impl of same ISA.
- M1 clock = 50 MHz, M2 clock = 75 MHz.
- M1 CPI = 2.8, M2 CPI = 3.2 for a program.
- How many times faster is M2 than M1?

$$\frac{\text{ExTime}_{M1}}{\text{ExTime}_{M2}} = \frac{\text{IC}_{M1} \times \text{CPI}_{M1} / \text{Clock Rate}_{M1}}{\text{IC}_{M2} \times \text{CPI}_{M2} / \text{Clock Rate}_{M2}} = \frac{2.8/50}{3.2/75} = 1.31$$

- What clock rate of M1 will give same execution time?

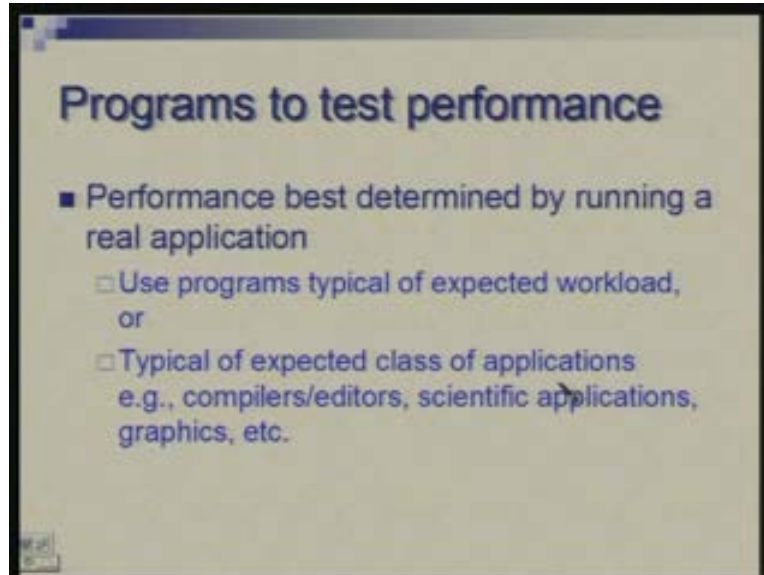
Again in this example we have two implementations of same instruction set architecture. So again instruction set is same these two are being called M1 and M2 they are different in clock 50 MHz in one case, 75 MHz in the second case, the CPI is also different 2.8 here and 3.2 here. So apparently what is happening here is that you have a longer clock cycle but more work is getting done in each cycle and here you have shorter cycles so less work is getting done in each cycle so you take more cycles here (Refer Slide Time: 21:44) and less cycles here.

Now how do you find the composite effect of these two; how do you compare?

The ratio of execution times of M1 and M2 would be the instruction count multiplied by CPI divided by clock rate of M1 and instruction count for M2 CPI of M2 and clock rate of M2. Now it is the instruction count which is common being the same instruction set architecture as we have seen earlier. So we can look at the remaining factors 2.8 by 50 and 3.2 by 75 which comes out to be 1.31 so it means that M1 takes longer as compared to M2 and the factor is that it is taking 31 percent more 31 percent longer than what M2 takes.

Now question could be asked in different ways also. So, if you had redesigned possibility for M1 and you were able to increase the clock rate keeping everything else the same keeping that CPI same then what clock rate for M1 would give you the same execution time? So here what you can do is you can keep this factor as unknown and put this as one (Refer Slide Time: 23:26) and that will give you the clock rate you require so I do not go through that calculation you can check it out.

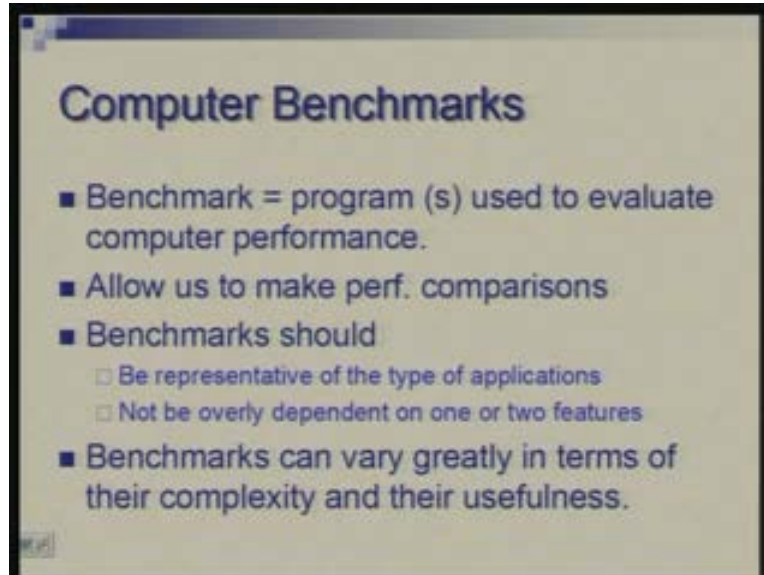
(Refer Slide Time: 23:36)



Now we have seen these examples where we try to see what happens if you change the design, what happens if you change the compiler. Similarly, you can examine the combined effect of both. So, in tutorial, probably we will take exercises which will try to look at more complex situations. But now we are faced with the question of which program you should use to check out the performance. If you are a very focused user you have a single program which you want to run over and over again then there is no problem; you can just see how long system works for that particular situation.

Or, **if you have** if you have not just one but a small set of programs or your work load typically some five six program you do over and over again you also may have an idea of how often you use program 1; how often you use program 2 and so on you could test for all of these and **get** take some kind of average. Depending upon your area of application, for example, **you could** you do compilations, editing, run some scientific application. So depending upon your area of application or your typical work load you can always work out. But the question is that can you do something in a more general sense. So this is a very personalized way of evaluating performance but are there general attempts; in general can you say A is better than B or so on. So, for that, one has what is called benchmarks.

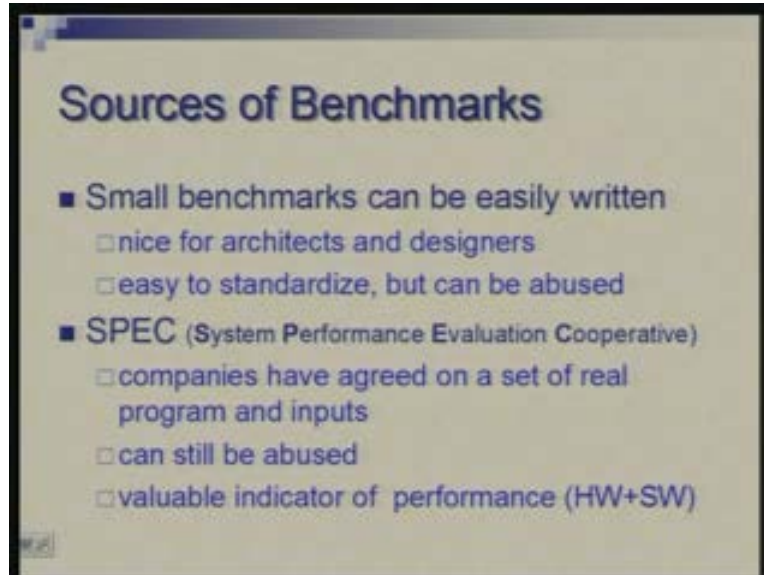
(Refer Slide Time: 25:34)



Benchmarks are basically programs used for testing the performance. It could be a single program or a set of programs which collectively are used for testing the performance. But one has to bear in mind that benchmark has to be or a set of benchmarks have to be appropriate for a user or a set of users. If a user runs commercial applications and benchmarks should be of that type; if user runs away the scientific application then benchmark should reflect that kind of application and a benchmark should be such that they test a given machine extensively; it should not be that they test some particular feature and a machine which is better in that feature but not better on the whole it would give a false improvement of..... false sense of related performance. So the benchmark should not be overly dependent upon a few set of features so that a machine vendor can also misrepresent the level of performance.

One can think of benchmarks of a varied complexity could be very small simple benchmarks and they could be very complex and accordingly their usefulness would vary.

(Refer Slide Time: 27:08)



So who defines these benchmarks?

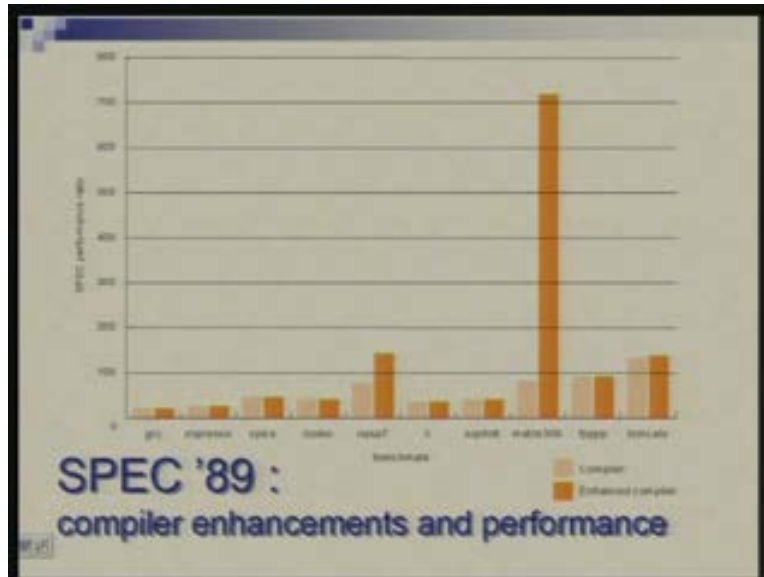
Benchmarks can be defined by users or the vendors or may be mutually agreed upon. Small benchmarks are easy to write and they are nice for discussing the performance by let us say designers and architects; **you can** you can closely see what is happening, how a benchmark is performing, where it is not performing and you can analyze it very nicely, they can be easily standardized but they can be abused.

For example, there have been cases of small benchmarks and cases where computer vendors would generate very highly optimized code for that particular benchmark so you could in fact have a switch in your compiler that if the code is this or just if the source is this then put this as the code and it could give again a false sense of high performance. So a consortium was developed in middle of 90s which is called SPEC and it stands for System Performance Evaluation Cooperative so there are number of industries that have joined hands to standardize this process of defining benchmarks, declaring performance, summarizing and tabulating the performance and bringing a common platform to this issue of performance evaluation.

So the attempt here is to take real programs, do not write small artificial or synthetic programs but take real programs which users use and standardize them, take some standard implementations, define some test inputs and so on and make them available on a common platform. So one could still abuse these because even in large programs what may happen is that lot of time gets spent on small pieces of code so if you could really attack those and make your machine work faster than those then things can go wrong but since **since** you are talking of not just one benchmark but a collection of benchmarks number of programs it may be somewhat harder to pick out the critical code of each of the programs and make a machine work faster, all of them.

So one could abuse but chances of such abuses are much less and these benchmarks are valuable indicators of performance of hardware plus software the total environment in which you use. That means **you have** you are talking of a machine a hardware and a compiler which generates a code because these benchmarks are not defined in assembly language of a particular machine; they are defined in standard high level languages and therefore role of a compiler is important. So what one is evaluating is not just the machine but compiler plus the machine combination.

(Refer Slide Time: 30:32)



This is a typical result of carrying out a set of benchmark tests for a given machine. Here you are showing comparison of benchmark figures for two different compilers and the same machine. **So this** this lighter color is for one compiler and this orange one is for second compiler which is an enhanced version of first one the improved version that does certain optimization and the text is actually very small here so you may be not able to read.

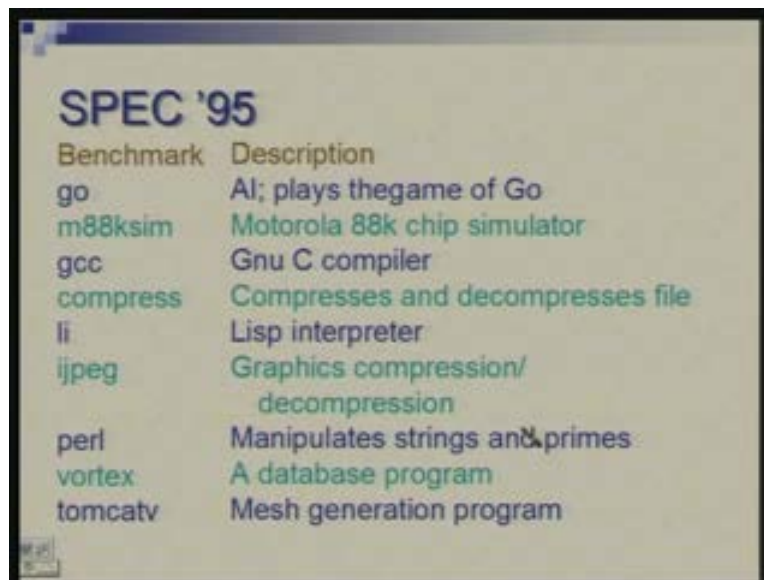
What we are showing on the vertical axis is the performance rating. So this is a number where higher number means higher performance and this number is obtained by running **running** the program, finding its time and taking reciprocal with a certain scaling factor so higher the number means higher is the performance or more is the execution time. And you would see a pair of bars; each pair corresponds to one particular benchmark so this one is for example gcc which is a C compiler so it is the time compiler takes to compile a program which we are actually measuring here.

So gcc compiler was compiled by this compiler and that compiler and the two times are different; there is a marginal improvement in the rating you do not see much difference here. this is a stress (Refer Slide Time: 32:14) over which is some circuit optimization program SPICE is a simulation program and so on and this is another, this is another, this is LISP interpreter.



So here you would see what has happened (Refer Slide Time: 32:39) that between light orange and dark orange there is a dramatic difference. So essentially what this compiler did was some clever trick that this matrix were into program, it could optimize some very crucial part of that program some loops and give a dramatic performance improvement. But if you look at in totality there is an improvement but not so dramatic. If you had looked at this in isolation you would have startling results.

(Refer Slide Time: 33:02)



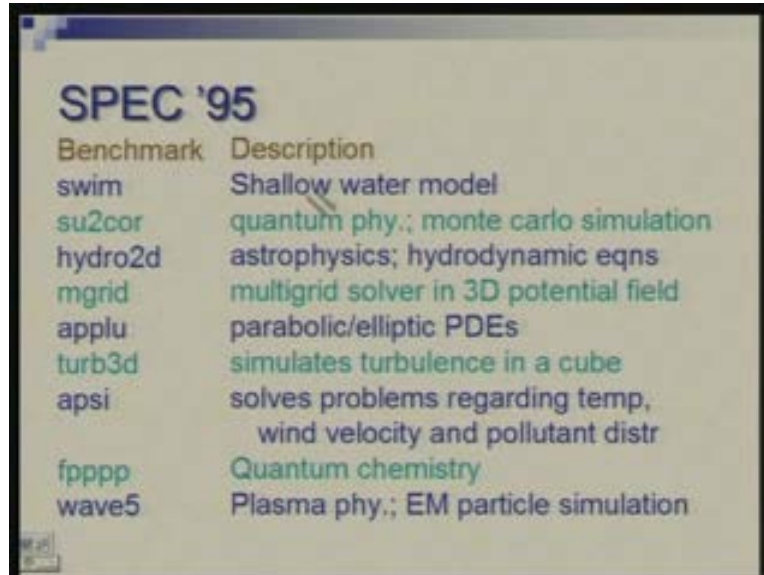
Benchmark	Description
go	AI; plays the game of Go
m88ksim	Motorola 88k chip simulator
gcc	Gnu C compiler
compress	Compresses and decompresses file
li	Lisp interpreter
jpeg	Graphics compression/ decompression
perl	Manipulates strings and primes
vortex	A database program
tomcatv	Mesh generation program

So what are these benchmarks?

Beginning with sometime around 95 a set of programs get designed a set of benchmarks. There are two sets actually those which are involved in integer computation so SPEC int and SPEC fp which is a floating-point. **so because you know** users working with floating-point heavy numerical computation they have different set of typical programs, those who work in symbolic computation or in non-numeric computation the set of benchmarks may be different. So in the integer case you have things like compiler, simulator, lisp, interpreter and so on. So here is a set of benchmarks so you will get some idea.

‘go’ is some artificial intelligence program plays the game of ‘go’; m88ksim is a Motorola 88k chip simulator so simulates a processor; gcc is gnu C compiler compresses a program for compression and decompression of files; li is lisp interpreter; jpeg - jpeg is graphics compression decompression; perl is a benchmark which is written in language perl it manipulates strings and prime numbers; vortex is a database program; tomcatv is a mesh generation program.

(Refer Slide Time: 34:42)

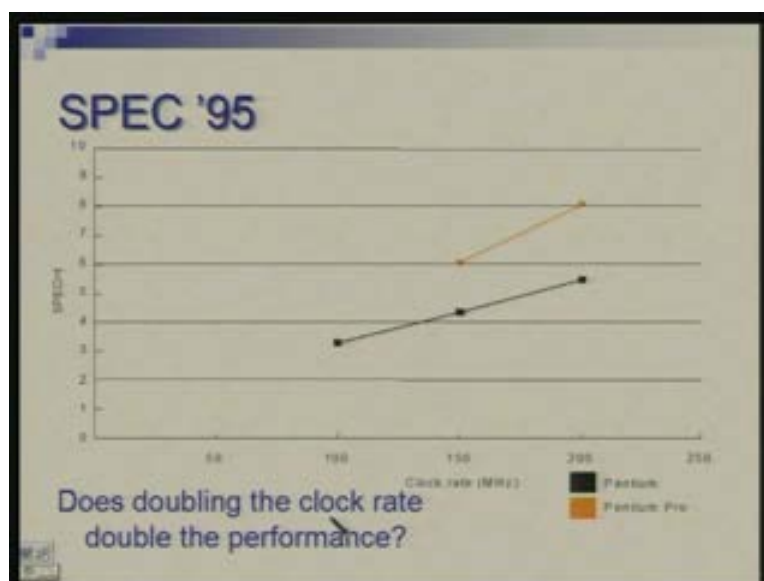


A slide titled 'SPEC '95' showing a list of benchmarks and their descriptions. The benchmarks are listed in two columns: 'Benchmark' and 'Description'. The benchmarks include swim, su2cor, hydro2d, mgrid, applu, turb3d, apsi, fpppp, and wave5.

Benchmark	Description
swim	Shallow water model
su2cor	quantum phy.; monte carlo simulation
hydro2d	astrophysics; hydrodynamic eqns
mgrid	multigrid solver in 3D potential field
applu	parabolic/elliptic PDEs
turb3d	simulates turbulence in a cube
apsi	solves problems regarding temp, wind velocity and pollutant distr
fpppp	Quantum chemistry
wave5	Plasma phy.; EM particle simulation

Swim is a shallow water model..... I am sorry these are not just these are actually mixture of integer and floating-point I can see both here. then there are benchmarks from quantum physics, astrophysics, 3D equations solver, partial differential equations something which simulates turbulence in a cube, solves problems regarding temperature, wind, velocity and pollutant distribution, this is from quantum chemistry (Refer Slide Time: 35:12), this is from plasma physics and so on.

(Refer Slide Time: 35:15)



So these benchmarks have been revised from time to time because as time changes the applications change, the machines change, they... the amount of memory which is

available in the machines which were at which were prevalent in 95 is not same as what is available today. so you need different programs, different set of programs to evaluate in benchmark content ready machines and therefore periodically these keep getting revised and it is always meaningful to work with the current set of benchmarks. So using these benchmarks you can answer a variety of questions.

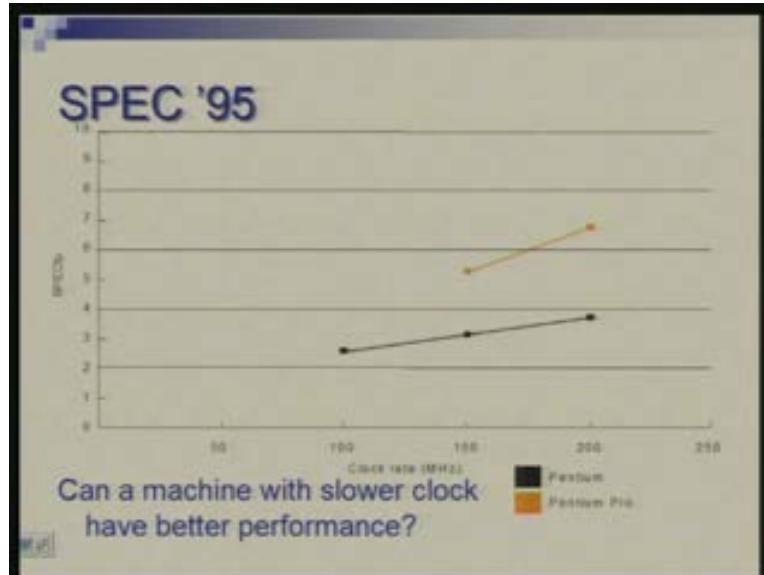
For example, you may take same machine and look at its two different versions with different clock rate and ask a question whether doubling the clock rate doubles the performance or not. So here (Refer Slide Time: 36:21) you run pent[ium] some program of SPEC int set over Pentium and Pentium pro so this refers to Pentium, this refers to Pentium pro; for each case you have done it with different machines, so, for Pentium it is 100 MHz, 150 MHz and 200 MHz and how much is the performance varying; here it is something like 3 a little over 3 and here it is little over 5 so the clock has been doubled from 100 MHz to 200 MHz. but as you can see this number has not doubled it is less than double.

So what could be the reason?

In our formula we had clock frequency of a proportional factor what happened to that [Conversation between student and Professor:.....(37:18)] okay suppose cycles per instruction it also same] suppose cycles per instruction is also same..... yeah, what has happened here is that our formula does not take into account the memory speed, we are assuming that there are no additional cycles for memory so either you have to count them separately or actually adjust them in CPI itself so you could have effective CPI which would try to take into account extra time which may get spent if memory is slow. So when we are doubling the processor frequency we have not said that have you doubled the memory speed also or not.

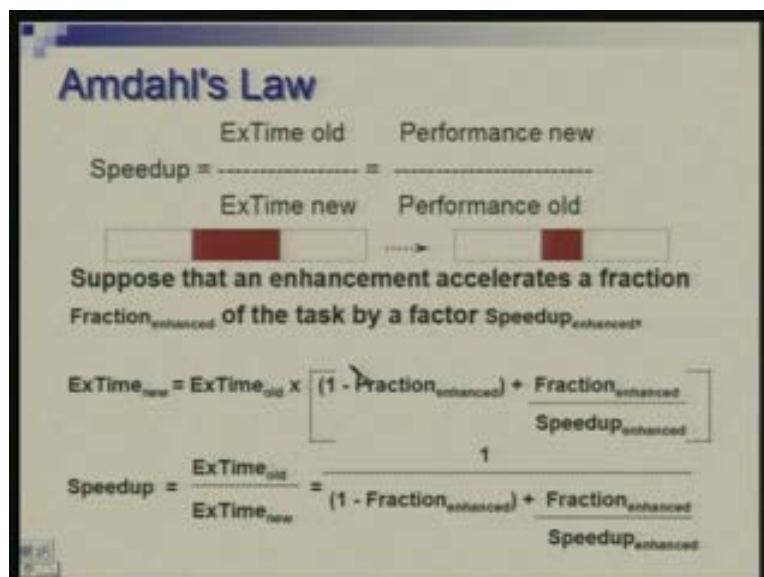
So we will look at memory effect later; this is just to remind you that we have looked we are looking at still in a very small domain just looking at processor. But when you account for other things: memory, IO, peripherals and so on I think it will become little more complex. But these benchmarks are run on physical systems so memory is there, IO is there, everything is there and you have run a program from end to end from input to its output.

(Refer Slide Time: 38:39)



So in this case, in this diagram the question which is being asked **so in this case in this diagram the question which is being asked** is can a machine with slower clock have a better performance. Again simple question so same two machines Pentium and Pentium pro same 3 points the benchmark we have taken are SPEC fp that means for floating-point performance. So here you put notice that Pentium pro with 150MHz is out performing Pentium with 200 MHz. So although they have same set of instructions Pentium pro has a few extra of course. But the way the instructions are implemented is different and therefore with the lower clock itself it can outperform.

(Refer Slide Time: 39:33)



We are actually concerned about speeding up execution of programs eventually by various means by improving the architecture, improving instruction set, improving the hardware implementation, improving the compiler the way code is generated but we must keep in mind, very simple but important law called Amdahl's law. So first let us define what is speedup.

You have [.....40:08] old execution time and you have hopefully reduced new execution time, you take the ratio and that factor gives you speedup, so simple. Or alternatively you could take the ratio of performance figures which are reciprocal of execution time so performance of new or performance of load.

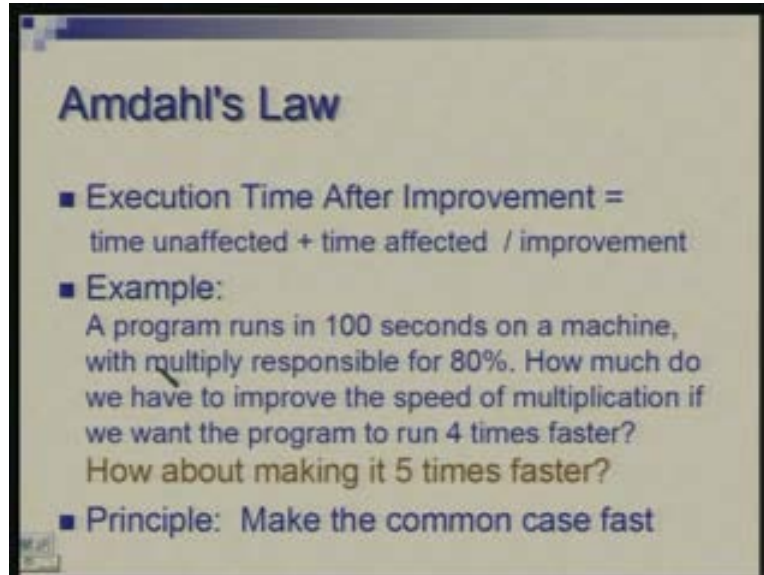
Now what happens is that quite often you may speedup only part of the program execution. So let us say this big rectangle represents the entire program or the time spent by entire program and your attention was focused on this shaded area. So it is this part you are focused on, you have designed your architecture or compiler so that this gets speeded up. Or there is a technique which you have in mind which affects only this part and does not affect rest of the part. So suppose this time shrinks to this by certain amount where the remaining part remains unchanged so what is the effect on the whole.

Let us say we tried to find figure out the fraction which is enhanced fraction which is not enhanced. So fraction subscript enhanced of the task is speeded up by a factor called speedup subscript enhanced. so this is a fraction, let us say you might say that I am taking three fourths of the program or 75 percent of the program and speeding it by a factor of 2 so these are the figures I am talking off; this is a fraction of the program you are speeding up and that is the factor by which you are speeding it up.

The new execution time can be now related to old execution time by this factor. So what you are saying is that 1 minus this fraction remains unchanged and the fraction which you have enhanced is reduced by this factor. So, if you had for example 75 percent of the program speeded up by a factor of 2 so this 0.25 remains unchanged and 0.75 divided by 2 is the contribution of that speeded up part. So the speed up would be the ratio of these two: execution time old and execution time new and what you will get is 1 upon this.

Now basically this is how the noise is captured and this law says that you have to keep in mind the part which is being speeded up and often you are not speeding up the whole thing. So the effect of this in the overall speed up may get limited by how big this fraction is. So if you are only taking of program and speeding, speeding it by a large factor still the total speedup cannot be more than 2 you get that. Suppose your attention is only on half of the program and you might infinitely speed it up but the overall time cannot be more than 2 the overall speedup cannot be more than 2 because the other half remains as it is. So, execution time after improvement is time which is unaffected that we take as it is and time which is affected is divided by the improvement factor.

(Refer Slide Time: 43:41)



**Amdahl's Law**

- Execution Time After Improvement =  
time unaffected + time affected / improvement
- Example:  
A program runs in 100 seconds on a machine, with multiply responsible for 80%. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?  
How about making it 5 times faster?
- Principle: Make the common case fast

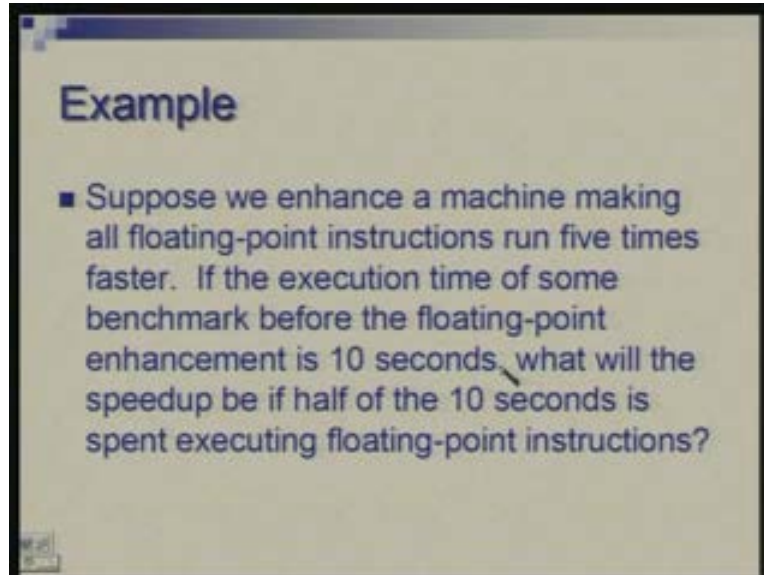
Therefore, now quantifying that with an example: Suppose the program runs in hundred seconds on a machine with a multiplication instruction responsible for 80 percent of the execution time so 80 percent of the time you are only multiplying and remaining 20 percent you may be adding, subtracting, loading data from memory, taking decision, branching and so on so it is a..... for whatever reason it is a multiply multiply or multiplication dominant program.

How much do we have to improve the speed of multiplication if you want to have the program run four times faster?

So from 100 seconds we want to reduce it to 25 seconds so what is the speedup factor required for multiplication alone? Pardon? [Conversation between student and Professor: (44:38)..... sixteen times] sixteen times, okay. So you can see that there is a huge improvement you need just to get four times improvement so your effort is to make it sixteen times but the net result you get is only four times improvement. so one Of course one principle which underlines all this is that you always have to find out which is the common part and that has to be made fast. So you have to pick up, well, 80 percentage of a good enough fraction here. Had this been even lower as I mentioned earlier 50 percent or if it is small fraction then your achievement is less. So you have to take something which is common and try to improve it as much as you can.



(Refer Slide Time: 45:30)



Another example: Suppose you want to take a machine and enhance its floating-point capability; suppose the floating-point instructions has speeded up five times, execution time of some benchmark before the floating-point enhancement is 10 seconds what will be the speedup if half of the 10 seconds is spent executing floating-point instructions. So now we are talking of a smaller fraction; only half the instructions are floating-point and we make them run five times faster. So basically you could say that this 10 seconds was divided into 5 and 5; 5 seconds are remaining unchanged, remaining 5 seconds are speeded up five times so that gives you 1 second so total time now is 6 seconds. So from 10 you have gone to 6 and therefore speedup is 10 by 6 or something like 1.67 so that is the factor by which you have speeded up.

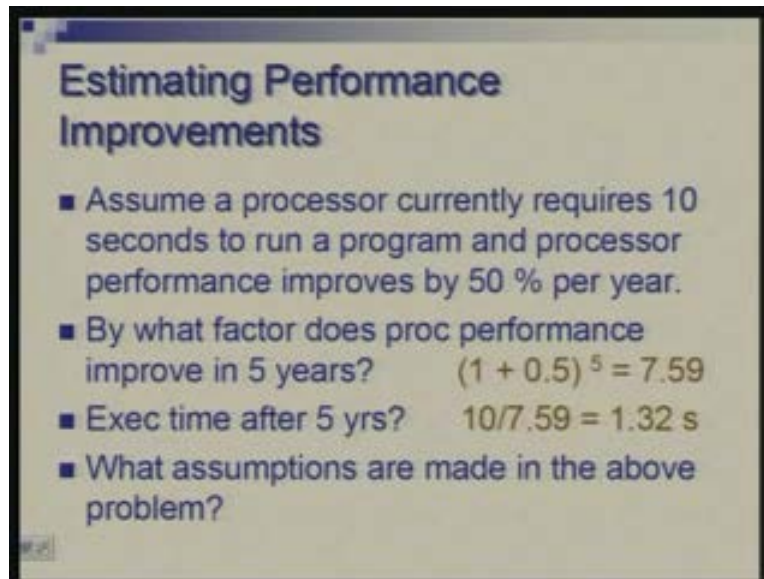
Again a question: Suppose we are looking for a benchmark to show off the new floating-point unit described above in the previous case and we want the overall benchmark to show a speedup of 3; one benchmark we are considering runs for 100 seconds with the old floating-point hardware. How much of the execution time would floating-point instructions have to account for in this program in order to yield the desired speedup on this benchmark?

The previous program where you had only 50 percent of instructions 50 percent of the time spent in floating-point will never show you a speedup of even 2 it will always be less than 2; in the previous case we got 1.67. But suppose you wanted to project your floating-point improvement what kind of benchmark you will have to choose? You want to choose a benchmark, you want to synthesize, you want to take up a benchmark which will show that with this floating-point improvement the whole thing runs three times faster. So now we are asking the question little differently.

So can you tell me what fraction of the original program should be a floating-point computation?

[Conversation between student and Professor: (48:12) 83 percent] Pardon 83 percent is that does everyone tally with that? You can work it out if you cannot quickly work it now.

(Refer Slide Time: 48:30)

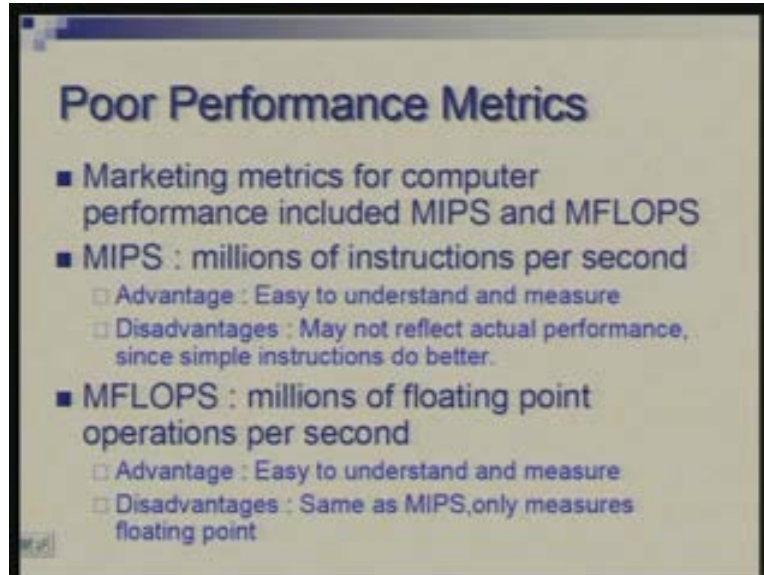


**Estimating Performance Improvements**

- Assume a processor currently requires 10 seconds to run a program and processor performance improves by 50 % per year.
- By what factor does proc performance improve in 5 years?  $(1 + 0.5)^5 = 7.59$
- Exec time after 5 yrs?  $10/7.59 = 1.32 \text{ s}$
- What assumptions are made in the above problem?

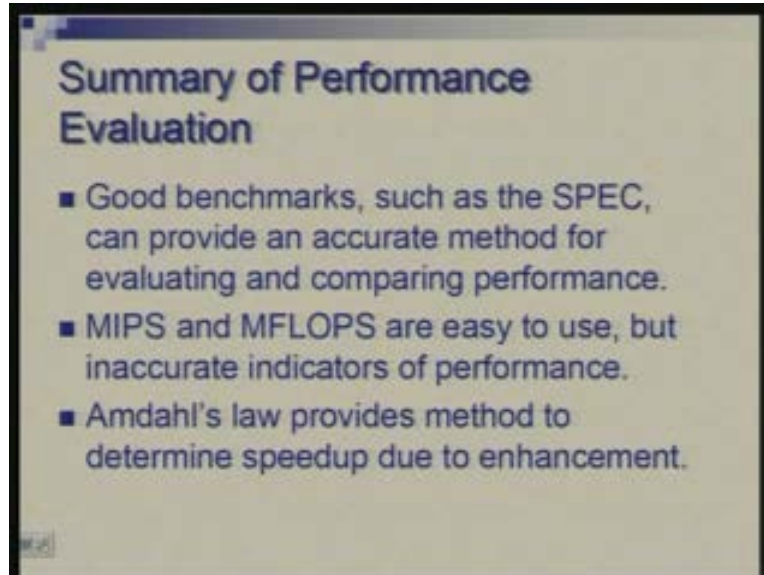
Here now **we have** let us look at improvement which takes place with technology, improvement with time. Assume a processor currently requires 10 seconds to run a program and a processor performance is improved by 50 percent every year. Let us say in simple terms you are getting x MHz today next day you will get 1.5 times x MHz, next time it will be 1.5 into 1.5 times x and so on. So every year you are seeing that the performance is going up by 50 percent. So by what factor performance improves in five years but it is very straightforward: 1 plus 0.5 raised to power 5 or you get a factor of 7.59 and execution time after five years would reduce by the same factor. So if it is 10 seconds it will become 10 by 7.59 or 1.32 seconds. So it is nice but can you spot what assumptions we have made what simplifying assumptions we have made in this calculation. Sorry? Yeah that is one of assumption and again let me again remind you that we have not said anything about the memory. Is memory improving by same factor or not; so here we have effectively assumed memory also improves by same factor if it does not then things will be different.

(Refer Slide Time: 50:20)



I have mentioned already about possible performance indicator which people have used is MIPS. Another one which has been used in past is MFLOPS or mega flops. So particularly for floating-point operations people have talked of MFLOPS which stands for million floating-point operations per second. So it is a metric again similar to MIPS which is talking of million instructions per second but here the focus is on floating-point which is somewhat relevant for people who do predominantly floating-point computation. But once again there is the other things which are there in the program it is not entirely floating-point. So advantage is with such measures is that they are easy to understand, measure and project but they are fallacies they may not be actually reflecting performance. Once again in tutorial we will take exercises which bring out the problem with these.

(Refer Slide Time: 50:20)

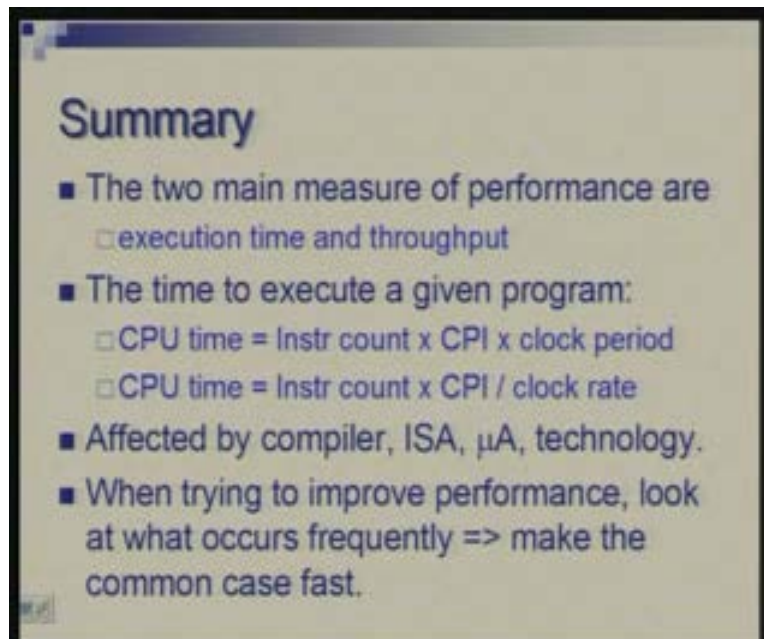


### Summary of Performance Evaluation

- Good benchmarks, such as the SPEC, can provide an accurate method for evaluating and comparing performance.
- MIPS and MFLOPS are easy to use, but inaccurate indicators of performance.
- Amdahl's law provides method to determine speedup due to enhancement.

So, to summarize what we have learnt that you need good benchmarks, standardized benchmarks which are available across the industries for evaluation and performance for such aspect. Measures like MIPS and mega flops look easy and simple but they could be misleading. And one has to keep in mind Amdahl's law while talking of speedup due to some enhancements.

(Refer Slide Time: 51:48)



### Summary

- The two main measure of performance are
  - execution time and throughput
- The time to execute a given program:
  - $\text{CPU time} = \text{Instr count} \times \text{CPI} \times \text{clock period}$
  - $\text{CPU time} = \text{Instr count} \times \text{CPI} / \text{clock rate}$
- Affected by compiler, ISA,  $\mu\text{A}$ , technology.
- When trying to improve performance, look at what occurs frequently => make the common case fast.

So now on the whole in the previous lecture and in this lecture we have seen that there are two measures of performance: execution time and throughput; we have focused on execution time which is given by a simple formula and these factors do get influenced by compiler technology, instruction set architecture, micro architecture which means how instructions are implemented in hardware and the basic circuit technology or fabrication technology. So when trying to improve performance we should try to capture a large fraction the common case and try to make it as fast as possible. Okay, I will stop with that.