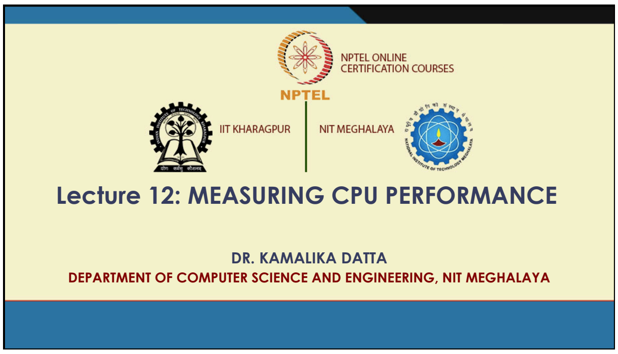


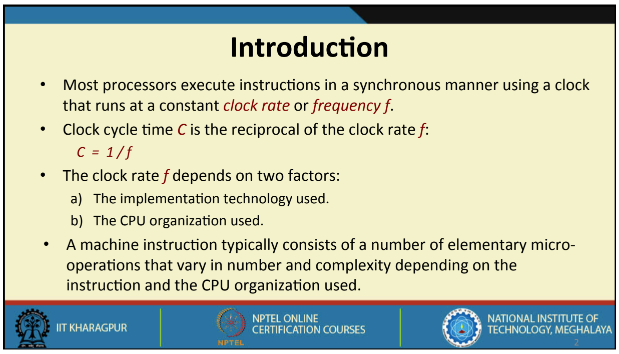
NPTEL ONLINE CERTIFICATION COURSES

NPTEL

IIT KHARAGPUR | NIT MEGHALAYA

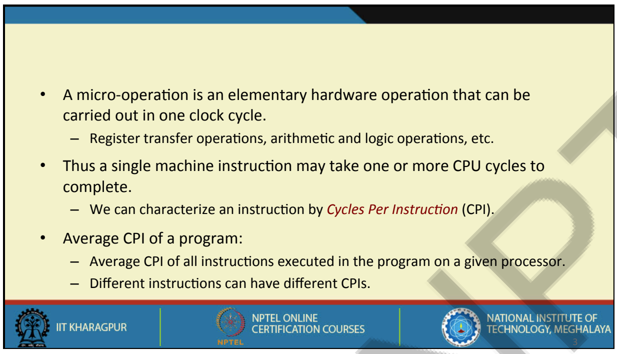
Lecture 12: MEASURING CPU PERFORMANCE

DR. KAMALIKA DATTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, NIT MEGHALAYA

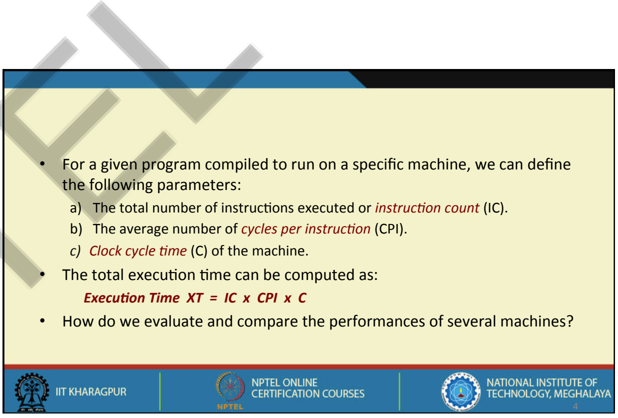


Introduction

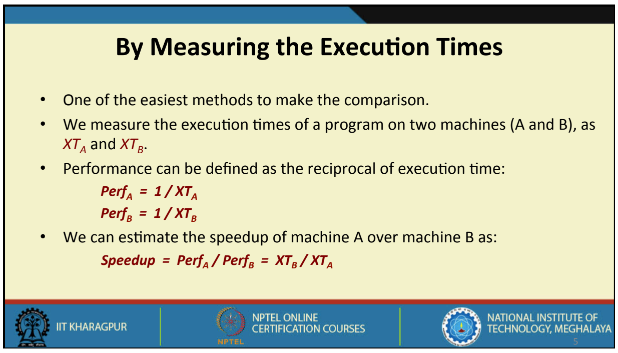
- Most processors execute instructions in a synchronous manner using a clock that runs at a constant *clock rate* or *frequency f*.
- Clock cycle time *C* is the reciprocal of the clock rate *f*:
  $C = 1/f$
- The clock rate *f* depends on two factors:
  a) The implementation technology used.
  b) The CPU organization used.
- A machine instruction typically consists of a number of elementary micro-operations that vary in number and complexity depending on the instruction and the CPU organization used.



- A micro-operation is an elementary hardware operation that can be carried out in one clock cycle.
  - Register transfer operations, arithmetic and logic operations, etc.
- Thus a single machine instruction may take one or more CPU cycles to complete.
  - We can characterize an instruction by *Cycles Per Instruction* (CPI).
- Average CPI of a program:
  - Average CPI of all instructions executed in the program on a given processor.
  - Different instructions can have different CPIs.

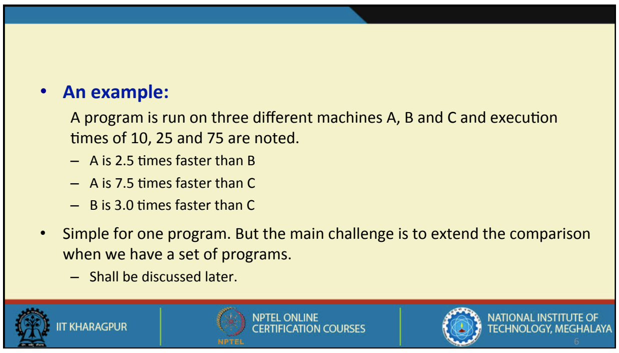


- For a given program compiled to run on a specific machine, we can define the following parameters:
  a) The total number of instructions executed or *instruction count* (IC).
  b) The average number of *cycles per instruction* (CPI).
  c) *Clock cycle time* (C) of the machine.
- The total execution time can be computed as:
  Execution Time XT = IC x CPI x C
- How do we evaluate and compare the performances of several machines?

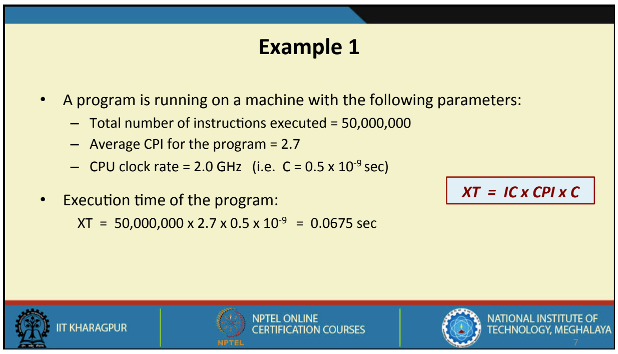


By Measuring the Execution Times

- One of the easiest methods to make the comparison.
- We measure the execution times of a program on two machines (A and B), as XT_A and XT_B.
- Performance can be defined as the reciprocal of execution time:
  $Perf_A = 1/XT_A$
  $Perf_B = 1/XT_B$
- We can estimate the speedup of machine A over machine B as:
  $Speedup = Perf_A / Perf_B = XT_B / XT_A$



- **An example:**
  A program is run on three different machines A, B and C and execution times of 10, 25 and 75 are noted.
  - A is 2.5 times faster than B
  - A is 7.5 times faster than C
  - B is 3.0 times faster than C
- Simple for one program. But the main challenge is to extend the comparison when we have a set of programs.
  - Shall be discussed later.

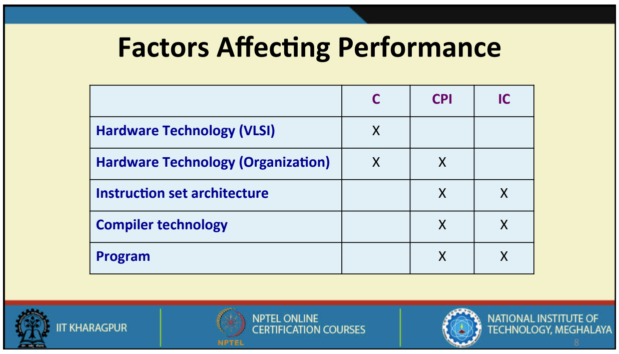


Example 1

- A program is running on a machine with the following parameters:
  - Total number of instructions executed = 50,000,000
  - Average CPI for the program = 2.7
  - CPU clock rate = 2.0 GHz (i.e. C = 0.5 x 10^{-9} sec)
- Execution time of the program:

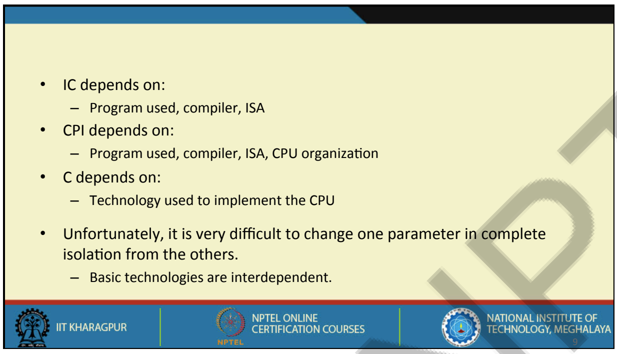
  XT = 50,000,000 x 2.7 x 0.5 x 10^{-9} = 0.0675 sec

XT = IC x CPI x C

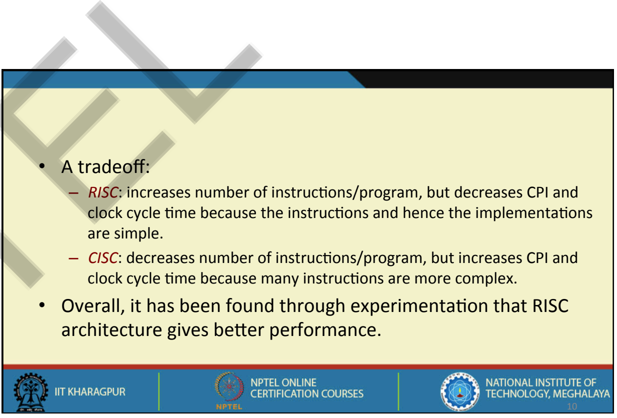


Factors Affecting Performance

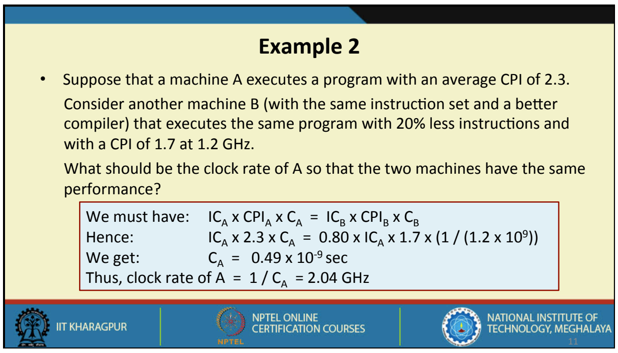
	C	CPI	IC
Hardware Technology (VLSI)	X		
Hardware Technology (Organization)	X	X	
Instruction set architecture		X	X
Compiler technology		X	X
Program		X	X



- IC depends on:
  - Program used, compiler, ISA
- CPI depends on:
  - Program used, compiler, ISA, CPU organization
- C depends on:
  - Technology used to implement the CPU
- Unfortunately, it is very difficult to change one parameter in complete isolation from the others.
  - Basic technologies are interdependent.



- A tradeoff:
  - *RISC*: increases number of instructions/program, but decreases CPI and clock cycle time because the instructions and hence the implementations are simple.
  - *CISC*: decreases number of instructions/program, but increases CPI and clock cycle time because many instructions are more complex.
- Overall, it has been found through experimentation that RISC architecture gives better performance.



Example 2

- Suppose that a machine A executes a program with an average CPI of 2.3.

  Consider another machine B (with the same instruction set and a better compiler) that executes the same program with 20% less instructions and with a CPI of 1.7 at 1.2 GHz.

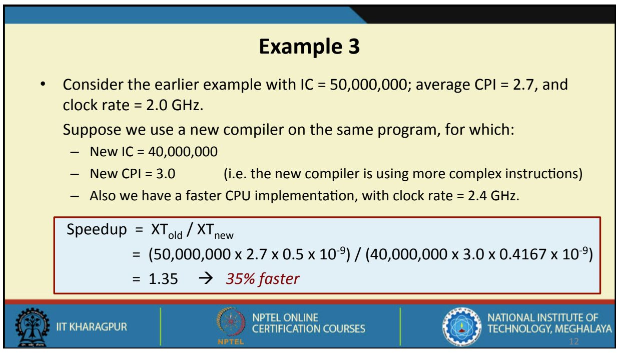
  What should be the clock rate of A so that the two machines have the same performance?

We must have: $IC_A \times CPI_A \times C_A = IC_B \times CPI_B \times C_B$

Hence: $IC_A \times 2.3 \times C_A = 0.80 \times IC_A \times 1.7 \times (1 / (1.2 \times 10^9))$

We get: $C_A = 0.49 \times 10^{-9}$ sec

Thus, clock rate of A = $1 / C_A$ = 2.04 GHz



Example 3

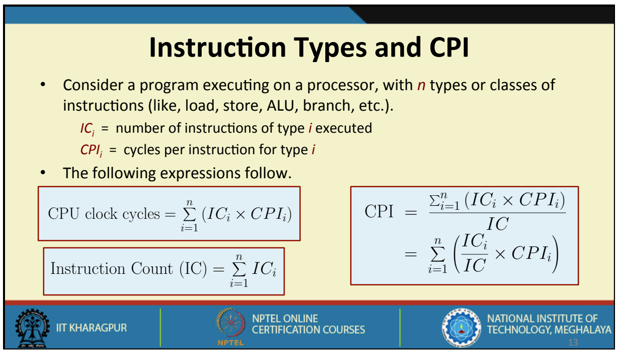
- Consider the earlier example with IC = 50,000,000; average CPI = 2.7, and clock rate = 2.0 GHz.

  Suppose we use a new compiler on the same program, for which:
  - New IC = 40,000,000
  - New CPI = 3.0 (i.e. the new compiler is using more complex instructions)
  - Also we have a faster CPU implementation, with clock rate = 2.4 GHz.

Speedup = XT_{old} / XT_{new}

= $(50{,}000{,}000 \times 2.7 \times 0.5 \times 10^{-9}) / (40{,}000{,}000 \times 3.0 \times 0.4167 \times 10^{-9})$

= 1.35 → *35% faster*



Instruction Types and CPI

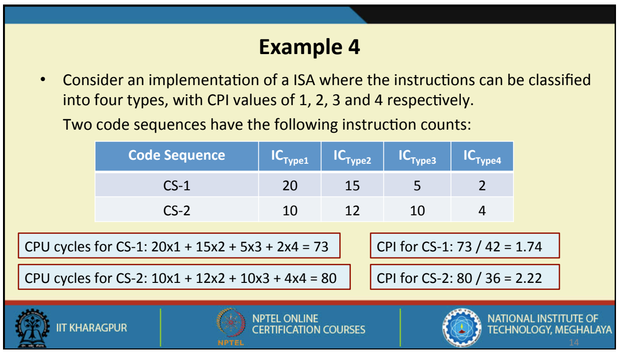
- Consider a program executing on a processor, with *n* types or classes of instructions (like, load, store, ALU, branch, etc.).
  - IC_i = number of instructions of type *i* executed
  - CPI_i = cycles per instruction for type *i*
- The following expressions follow.

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (IC_i \times CPI_i)$$

$$\text{Instruction Count (IC)} = \sum_{i=1}^{n} IC_i$$

$$\text{CPI} = \frac{\sum_{i=1}^{n} (IC_i \times CPI_i)}{IC} = \sum_{i=1}^{n} \left(\frac{IC_i}{IC} \times CPI_i\right)$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA | 13



Example 4

- Consider an implementation of a ISA where the instructions can be classified into four types, with CPI values of 1, 2, 3 and 4 respectively.

  Two code sequences have the following instruction counts:

Code Sequence	IC_{Type1}	IC_{Type2}	IC_{Type3}	IC_{Type4}
CS-1	20	15	5	2
CS-2	10	12	10	4

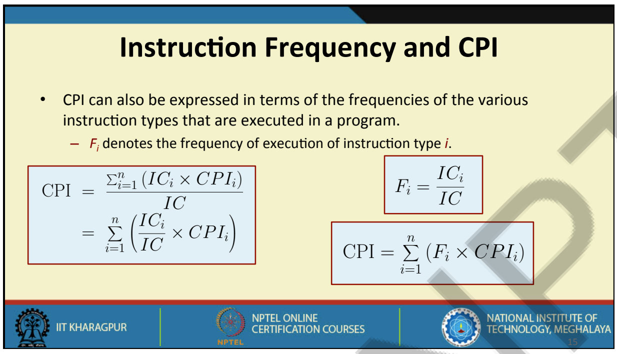
CPU cycles for CS-1: 20x1 + 15x2 + 5x3 + 2x4 = 73

CPI for CS-1: 73 / 42 = 1.74

CPU cycles for CS-2: 10x1 + 12x2 + 10x3 + 4x4 = 80

CPI for CS-2: 80 / 36 = 2.22

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA | 14



Instruction Frequency and CPI

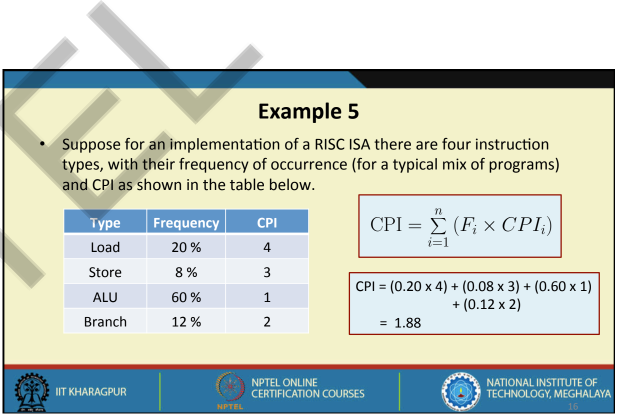
- CPI can also be expressed in terms of the frequencies of the various instruction types that are executed in a program.
  - F_i denotes the frequency of execution of instruction type *i*.

$$\text{CPI} = \frac{\sum_{i=1}^{n} (IC_i \times CPI_i)}{IC} = \sum_{i=1}^{n} \left(\frac{IC_i}{IC} \times CPI_i\right)$$

$$F_i = \frac{IC_i}{IC}$$

$$\text{CPI} = \sum_{i=1}^{n} (F_i \times CPI_i)$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA | 15



Example 5

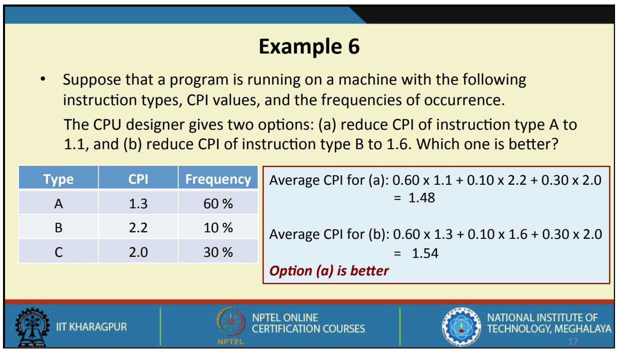
- Suppose for an implementation of a RISC ISA there are four instruction types, with their frequency of occurrence (for a typical mix of programs) and CPI as shown in the table below.

Type	Frequency	CPI
Load	20 %	4
Store	8 %	3
ALU	60 %	1
Branch	12 %	2

$$\text{CPI} = \sum_{i=1}^{n} (F_i \times CPI_i)$$

CPI = (0.20 x 4) + (0.08 x 3) + (0.60 x 1) + (0.12 x 2)
= 1.88

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA | 16



Example 6

- Suppose that a program is running on a machine with the following instruction types, CPI values, and the frequencies of occurrence.

  The CPU designer gives two options: (a) reduce CPI of instruction type A to 1.1, and (b) reduce CPI of instruction type B to 1.6. Which one is better?

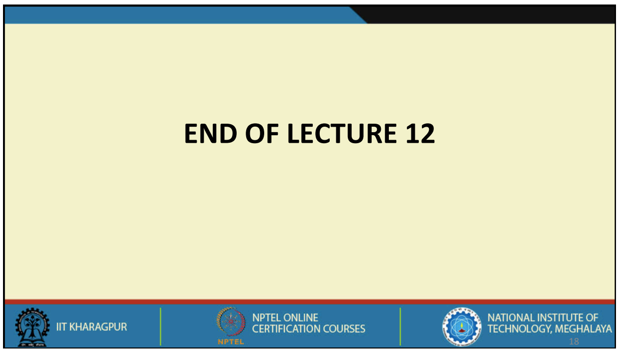
Type	CPI	Frequency
A	1.3	60 %
B	2.2	10 %
C	2.0	30 %

Average CPI for (a): 0.60 x 1.1 + 0.10 x 2.2 + 0.30 x 2.0
= 1.48

Average CPI for (b): 0.60 x 1.3 + 0.10 x 1.6 + 0.30 x 2.0
= 1.54

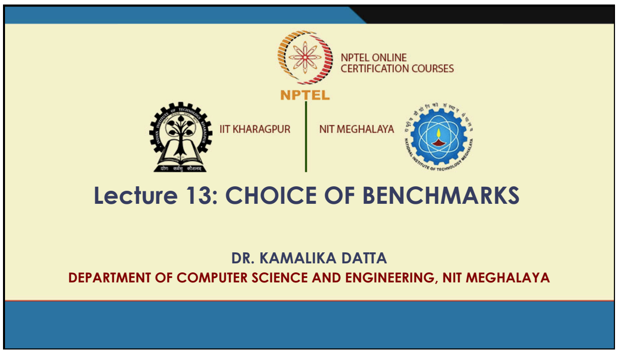
Option (a) is better

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA | 17



END OF LECTURE 12

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA | 18



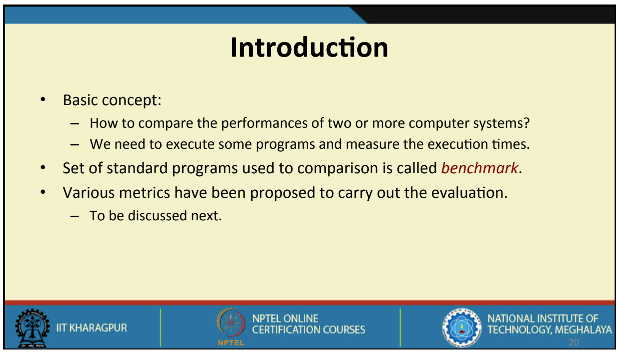
NPTEL ONLINE CERTIFICATION COURSES

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 13: CHOICE OF BENCHMARKS

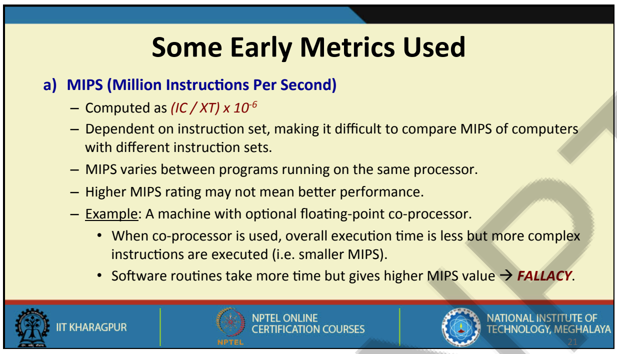
DR. KAMALIKA DATTA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, NIT MEGHALAYA



Introduction

- Basic concept:
  – How to compare the performances of two or more computer systems?
  – We need to execute some programs and measure the execution times.
- Set of standard programs used to comparison is called *benchmark*.
- Various metrics have been proposed to carry out the evaluation.
  – To be discussed next.



Some Early Metrics Used

a) MIPS (Million Instructions Per Second)

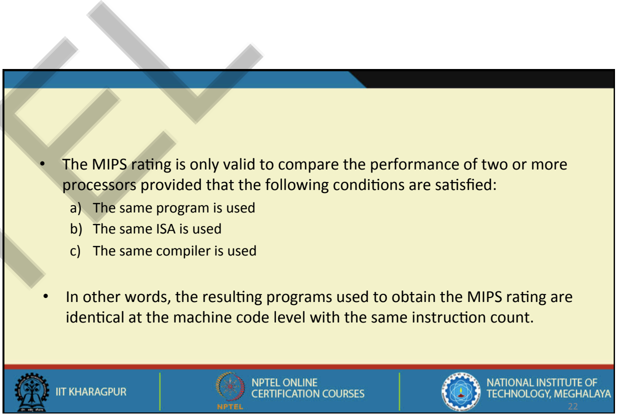
– Computed as *$(IC / XT) \times 10^{-6}$*

– Dependent on instruction set, making it difficult to compare MIPS of computers with different instruction sets.

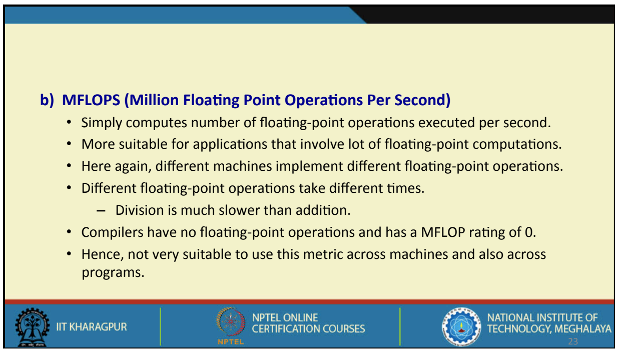
– MIPS varies between programs running on the same processor.

– Higher MIPS rating may not mean better performance.

– Example: A machine with optional floating-point co-processor.
  - When co-processor is used, overall execution time is less but more complex instructions are executed (i.e. smaller MIPS).
  - Software routines take more time but gives higher MIPS value → ***FALLACY***.

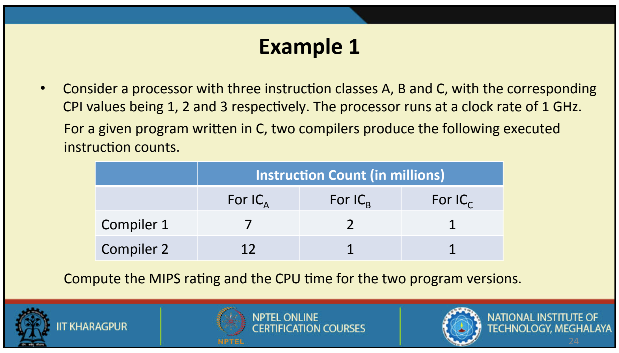


- The MIPS rating is only valid to compare the performance of two or more processors provided that the following conditions are satisfied:
  a) The same program is used
  b) The same ISA is used
  c) The same compiler is used
- In other words, the resulting programs used to obtain the MIPS rating are identical at the machine code level with the same instruction count.



b) MFLOPS (Million Floating Point Operations Per Second)

- Simply computes number of floating-point operations executed per second.
- More suitable for applications that involve lot of floating-point computations.
- Here again, different machines implement different floating-point operations.
- Different floating-point operations take different times.
  – Division is much slower than addition.
- Compilers have no floating-point operations and has a MFLOP rating of 0.
- Hence, not very suitable to use this metric across machines and also across programs.



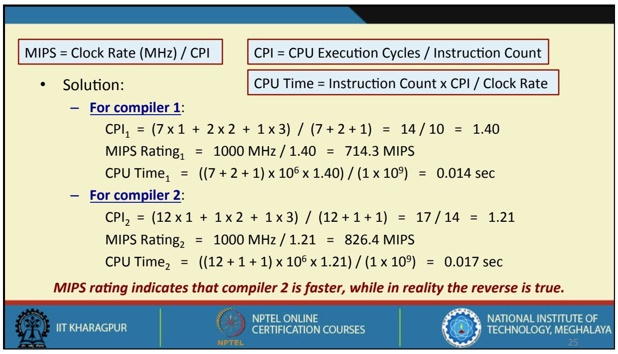
Example 1

- Consider a processor with three instruction classes A, B and C, with the corresponding CPI values being 1, 2 and 3 respectively. The processor runs at a clock rate of 1 GHz.

  For a given program written in C, two compilers produce the following executed instruction counts.

	Instruction Count (in millions)		
	For IC_A	For IC_B	For IC_C
Compiler 1	7	2	1
Compiler 2	12	1	1

Compute the MIPS rating and the CPU time for the two program versions.



MIPS = Clock Rate (MHz) / CPI

CPI = CPU Execution Cycles / Instruction Count

CPU Time = Instruction Count x CPI / Clock Rate

- Solution:
  - **For compiler 1**:

    $\text{CPI}_1 = (7 \times 1 + 2 \times 2 + 1 \times 3) / (7 + 2 + 1) = 14 / 10 = 1.40$

    $\text{MIPS Rating}_1 = 1000 \text{ MHz} / 1.40 = 714.3 \text{ MIPS}$

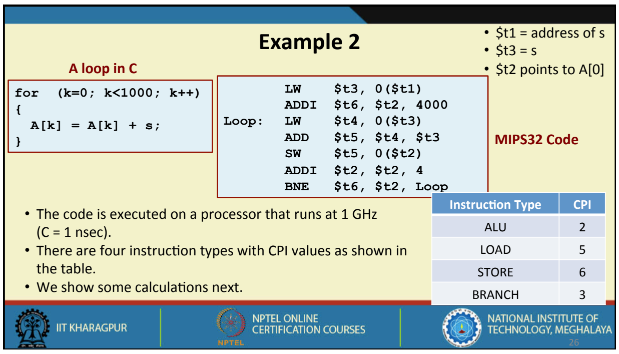
    $\text{CPU Time}_1 = ((7 + 2 + 1) \times 10^6 \times 1.40) / (1 \times 10^9) = 0.014 \text{ sec}$
  - **For compiler 2**:

    $\text{CPI}_2 = (12 \times 1 + 1 \times 2 + 1 \times 3) / (12 + 1 + 1) = 17 / 14 = 1.21$

    $\text{MIPS Rating}_2 = 1000 \text{ MHz} / 1.21 = 826.4 \text{ MIPS}$

    $\text{CPU Time}_2 = ((12 + 1 + 1) \times 10^6 \times 1.21) / (1 \times 10^9) = 0.017 \text{ sec}$

MIPS rating indicates that compiler 2 is faster, while in reality the reverse is true.



Example 2

- \$t1 = address of s
- \$t3 = s
- \$t2 points to A[0]

A loop in C

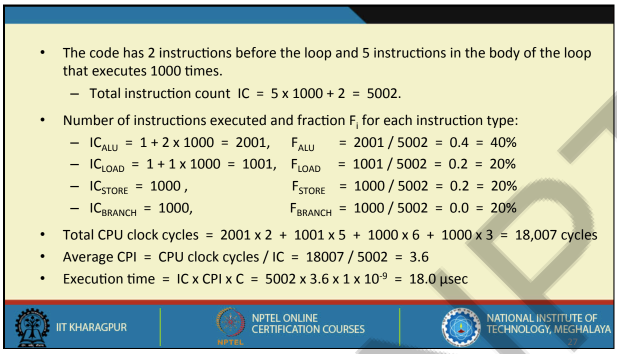
```
for  (k=0; k<1000; k++)
{
  A[k] = A[k] + s;
}
```

```
        LW    $t3, 0($t1)
        ADDI  $t6, $t2, 4000
Loop:   LW    $t4, 0($t3)
        ADD   $t5, $t4, $t3
        SW    $t5, 0($t2)
        ADDI  $t2, $t2, 4
        BNE   $t6, $t2, Loop
```

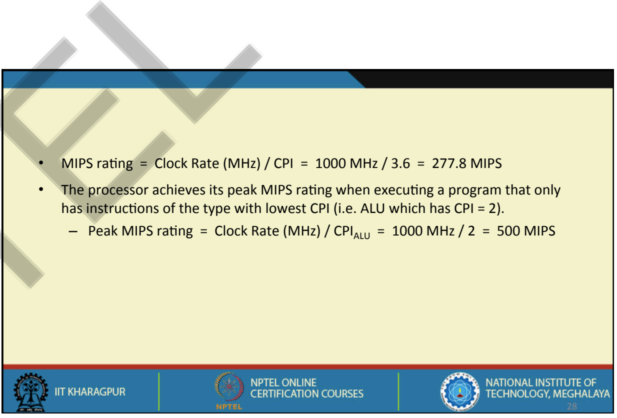
MIPS32 Code

- The code is executed on a processor that runs at 1 GHz (C = 1 nsec).
- There are four instruction types with CPI values as shown in the table.
- We show some calculations next.

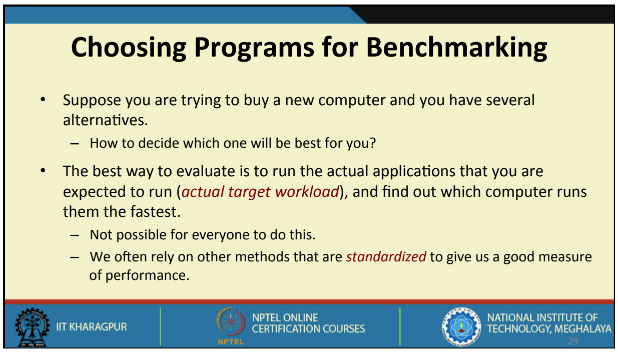
Instruction Type	CPI
ALU	2
LOAD	5
STORE	6
BRANCH	3



- The code has 2 instructions before the loop and 5 instructions in the body of the loop that executes 1000 times.
  - Total instruction count IC = 5 x 1000 + 2 = 5002.
- Number of instructions executed and fraction F_i for each instruction type:
  - $IC_{ALU} = 1 + 2 \times 1000 = 2001$, $F_{ALU} = 2001 / 5002 = 0.4 = 40\%$
  - $IC_{LOAD} = 1 + 1 \times 1000 = 1001$, $F_{LOAD} = 1001 / 5002 = 0.2 = 20\%$
  - $IC_{STORE} = 1000$, $F_{STORE} = 1000 / 5002 = 0.2 = 20\%$
  - $IC_{BRANCH} = 1000$, $F_{BRANCH} = 1000 / 5002 = 0.0 = 20\%$
- Total CPU clock cycles = 2001 x 2 + 1001 x 5 + 1000 x 6 + 1000 x 3 = 18,007 cycles
- Average CPI = CPU clock cycles / IC = 18007 / 5002 = 3.6
- Execution time = IC x CPI x C = $5002 \times 3.6 \times 1 \times 10^{-9}$ = 18.0 µsec

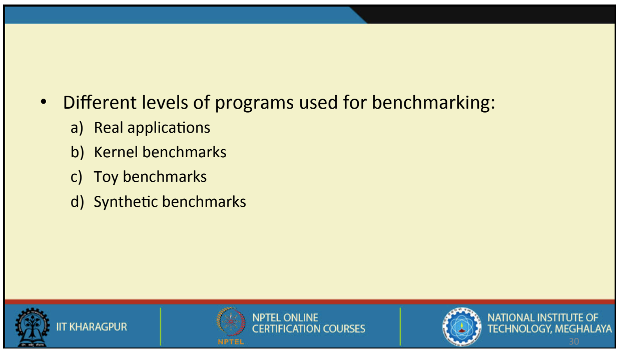


- MIPS rating = Clock Rate (MHz) / CPI = 1000 MHz / 3.6 = 277.8 MIPS
- The processor achieves its peak MIPS rating when executing a program that only has instructions of the type with lowest CPI (i.e. ALU which has CPI = 2).
  - Peak MIPS rating = Clock Rate (MHz) / CPI_{ALU} = 1000 MHz / 2 = 500 MIPS

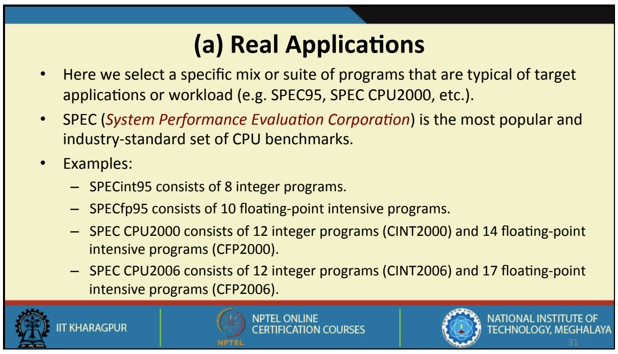


Choosing Programs for Benchmarking

- Suppose you are trying to buy a new computer and you have several alternatives.
  - How to decide which one will be best for you?
- The best way to evaluate is to run the actual applications that you are expected to run (*actual target workload*), and find out which computer runs them the fastest.
  - Not possible for everyone to do this.
  - We often rely on other methods that are *standardized* to give us a good measure of performance.

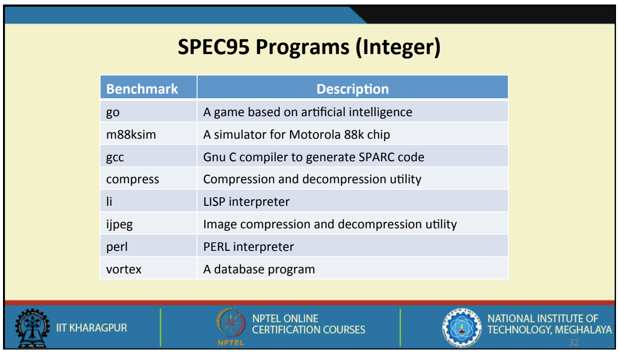


- Different levels of programs used for benchmarking:
  a) Real applications
  b) Kernel benchmarks
  c) Toy benchmarks
  d) Synthetic benchmarks



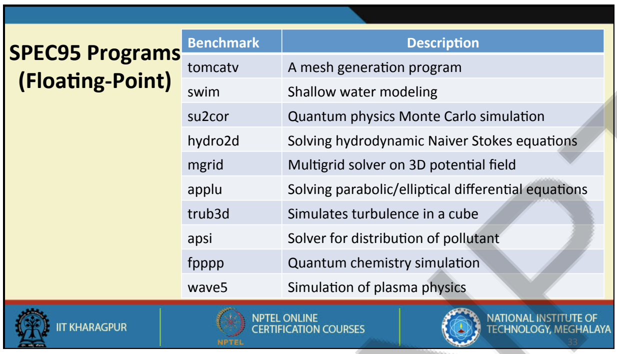
(a) Real Applications

- Here we select a specific mix or suite of programs that are typical of target applications or workload (e.g. SPEC95, SPEC CPU2000, etc.).
- SPEC (*System Performance Evaluation Corporation*) is the most popular and industry-standard set of CPU benchmarks.
- Examples:
  - SPECint95 consists of 8 integer programs.
  - SPECfp95 consists of 10 floating-point intensive programs.
  - SPEC CPU2000 consists of 12 integer programs (CINT2000) and 14 floating-point intensive programs (CFP2000).
  - SPEC CPU2006 consists of 12 integer programs (CINT2006) and 17 floating-point intensive programs (CFP2006).



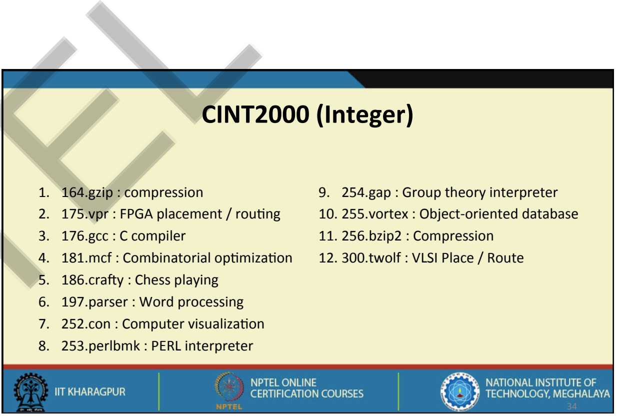
SPEC95 Programs (Integer)

Benchmark	Description
go	A game based on artificial intelligence
m88ksim	A simulator for Motorola 88k chip
gcc	Gnu C compiler to generate SPARC code
compress	Compression and decompression utility
li	LISP interpreter
ijpeg	Image compression and decompression utility
perl	PERL interpreter
vortex	A database program



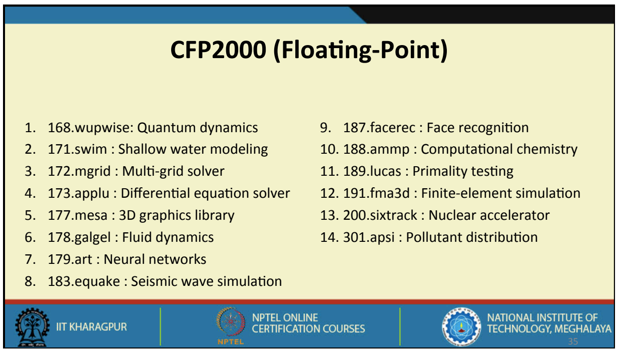
SPEC95 Programs (Floating-Point)

Benchmark	Description
tomcatv	A mesh generation program
swim	Shallow water modeling
su2cor	Quantum physics Monte Carlo simulation
hydro2d	Solving hydrodynamic Naiver Stokes equations
mgrid	Multigrid solver on 3D potential field
applu	Solving parabolic/elliptical differential equations
trub3d	Simulates turbulence in a cube
apsi	Solver for distribution of pollutant
fpppp	Quantum chemistry simulation
wave5	Simulation of plasma physics



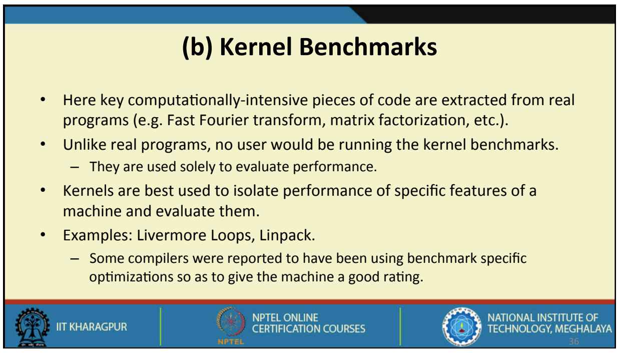
CINT2000 (Integer)

1. 164.gzip : compression
2. 175.vpr : FPGA placement / routing
3. 176.gcc : C compiler
4. 181.mcf : Combinatorial optimization
5. 186.crafty : Chess playing
6. 197.parser : Word processing
7. 252.con : Computer visualization
8. 253.perlbmk : PERL interpreter
9. 254.gap : Group theory interpreter
10. 255.vortex : Object-oriented database
11. 256.bzip2 : Compression
12. 300.twolf : VLSI Place / Route



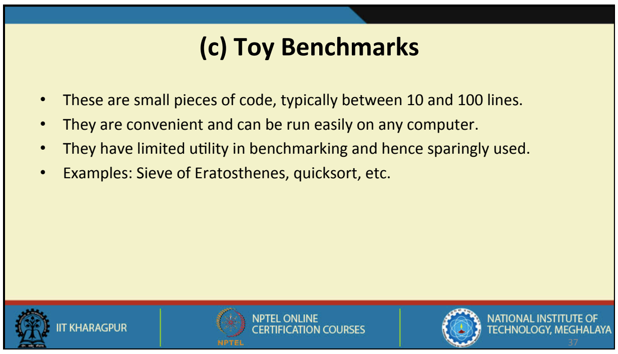
CFP2000 (Floating-Point)

1. 168.wupwise: Quantum dynamics
2. 171.swim : Shallow water modeling
3. 172.mgrid : Multi-grid solver
4. 173.applu : Differential equation solver
5. 177.mesa : 3D graphics library
6. 178.galgel : Fluid dynamics
7. 179.art : Neural networks
8. 183.equake : Seismic wave simulation
9. 187.facerec : Face recognition
10. 188.ammp : Computational chemistry
11. 189.lucas : Primality testing
12. 191.fma3d : Finite-element simulation
13. 200.sixtrack : Nuclear accelerator
14. 301.apsi : Pollutant distribution



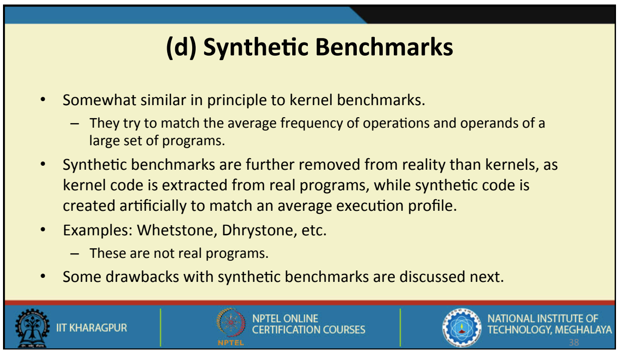
(b) Kernel Benchmarks

- Here key computationally-intensive pieces of code are extracted from real programs (e.g. Fast Fourier transform, matrix factorization, etc.).
- Unlike real programs, no user would be running the kernel benchmarks.
  - They are used solely to evaluate performance.
- Kernels are best used to isolate performance of specific features of a machine and evaluate them.
- Examples: Livermore Loops, Linpack.
  - Some compilers were reported to have been using benchmark specific optimizations so as to give the machine a good rating.



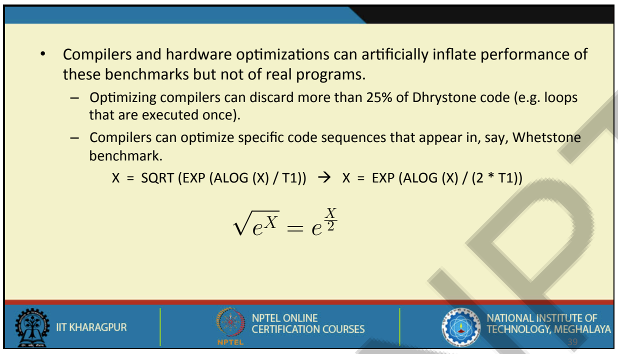
(c) Toy Benchmarks

- These are small pieces of code, typically between 10 and 100 lines.
- They are convenient and can be run easily on any computer.
- They have limited utility in benchmarking and hence sparingly used.
- Examples: Sieve of Eratosthenes, quicksort, etc.



(d) Synthetic Benchmarks

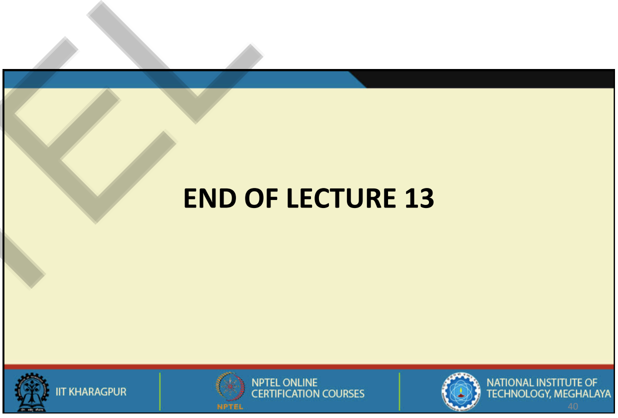
- Somewhat similar in principle to kernel benchmarks.
  - They try to match the average frequency of operations and operands of a large set of programs.
- Synthetic benchmarks are further removed from reality than kernels, as kernel code is extracted from real programs, while synthetic code is created artificially to match an average execution profile.
- Examples: Whetstone, Dhrystone, etc.
  - These are not real programs.
- Some drawbacks with synthetic benchmarks are discussed next.



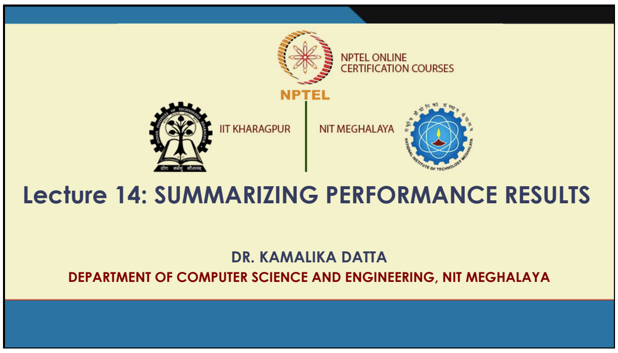
- Compilers and hardware optimizations can artificially inflate performance of these benchmarks but not of real programs.
  - Optimizing compilers can discard more than 25% of Dhrystone code (e.g. loops that are executed once).
  - Compilers can optimize specific code sequences that appear in, say, Whetstone benchmark.

X = SQRT (EXP (ALOG (X) / T1)) → X = EXP (ALOG (X) / (2 * T1))

$$\sqrt{e^X} = e^{\frac{X}{2}}$$



END OF LECTURE 13



NPTEL ONLINE CERTIFICATION COURSES

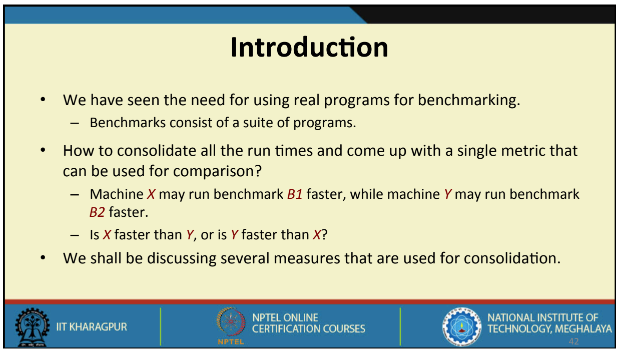
NPTEL

IIT KHARAGPUR NIT MEGHALAYA

Lecture 14: SUMMARIZING PERFORMANCE RESULTS

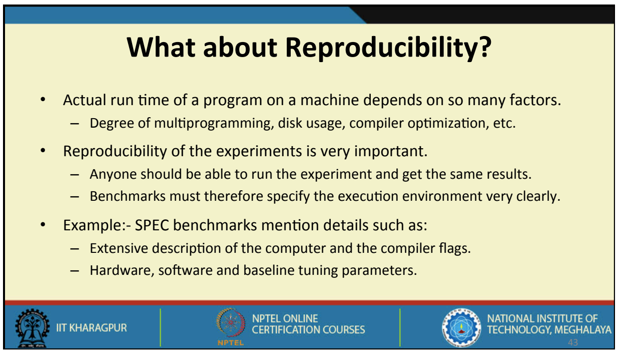
DR. KAMALIKA DATTA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, NIT MEGHALAYA



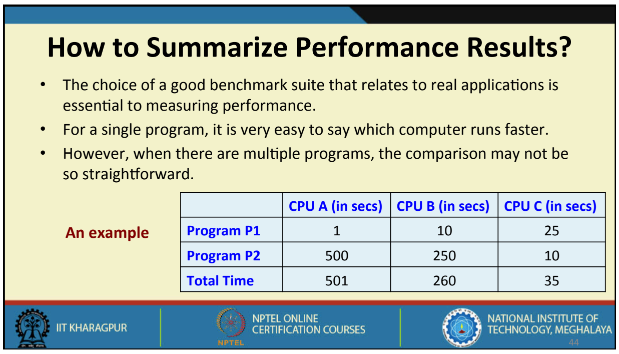
Introduction

- We have seen the need for using real programs for benchmarking.
  - Benchmarks consist of a suite of programs.
- How to consolidate all the run times and come up with a single metric that can be used for comparison?
  - Machine *X* may run benchmark *B1* faster, while machine *Y* may run benchmark *B2* faster.
  - Is *X* faster than *Y*, or is *Y* faster than *X*?
- We shall be discussing several measures that are used for consolidation.



What about Reproducibility?

- Actual run time of a program on a machine depends on so many factors.
  - Degree of multiprogramming, disk usage, compiler optimization, etc.
- Reproducibility of the experiments is very important.
  - Anyone should be able to run the experiment and get the same results.
  - Benchmarks must therefore specify the execution environment very clearly.
- Example:- SPEC benchmarks mention details such as:
  - Extensive description of the computer and the compiler flags.
  - Hardware, software and baseline tuning parameters.

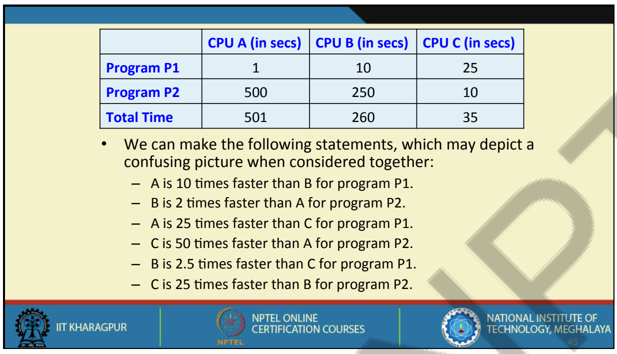


How to Summarize Performance Results?

- The choice of a good benchmark suite that relates to real applications is essential to measuring performance.
- For a single program, it is very easy to say which computer runs faster.
- However, when there are multiple programs, the comparison may not be so straightforward.

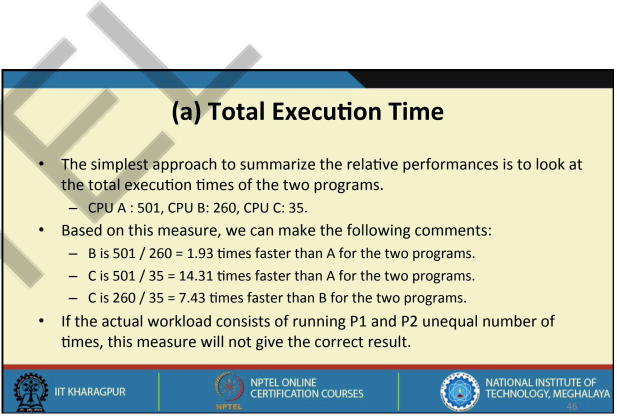
An example

	CPU A (in secs)	CPU B (in secs)	CPU C (in secs)
Program P1	1	10	25
Program P2	500	250	10
Total Time	501	260	35



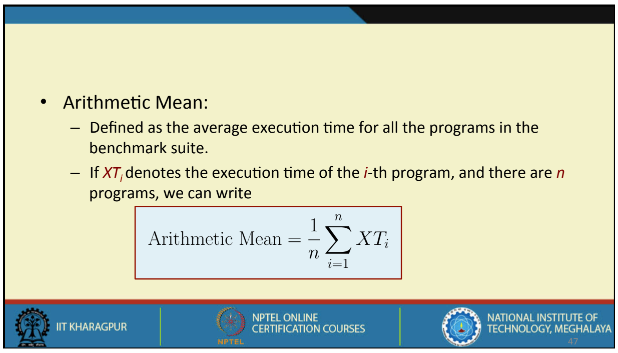
	CPU A (in secs)	CPU B (in secs)	CPU C (in secs)
Program P1	1	10	25
Program P2	500	250	10
Total Time	501	260	35

- We can make the following statements, which may depict a confusing picture when considered together:
  - A is 10 times faster than B for program P1.
  - B is 2 times faster than A for program P2.
  - A is 25 times faster than C for program P1.
  - C is 50 times faster than A for program P2.
  - B is 2.5 times faster than C for program P1.
  - C is 25 times faster than B for program P2.



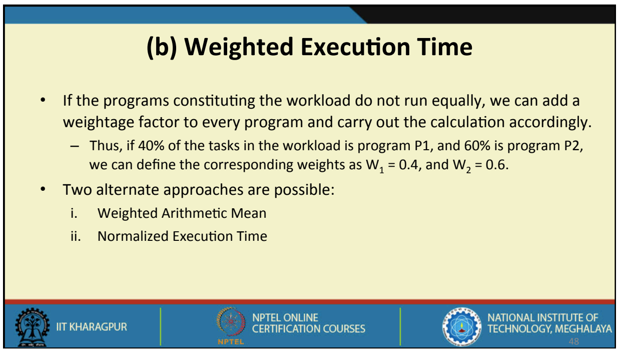
(a) Total Execution Time

- The simplest approach to summarize the relative performances is to look at the total execution times of the two programs.
  - CPU A : 501, CPU B: 260, CPU C: 35.
- Based on this measure, we can make the following comments:
  - B is 501 / 260 = 1.93 times faster than A for the two programs.
  - C is 501 / 35 = 14.31 times faster than A for the two programs.
  - C is 260 / 35 = 7.43 times faster than B for the two programs.
- If the actual workload consists of running P1 and P2 unequal number of times, this measure will not give the correct result.



- Arithmetic Mean:
  - Defined as the average execution time for all the programs in the benchmark suite.
  - If XT_i denotes the execution time of the i-th program, and there are n programs, we can write

$$\boxed{\text{Arithmetic Mean} = \frac{1}{n}\sum_{i=1}^{n} XT_i}$$



(b) Weighted Execution Time

- If the programs constituting the workload do not run equally, we can add a weightage factor to every program and carry out the calculation accordingly.
  - Thus, if 40% of the tasks in the workload is program P1, and 60% is program P2, we can define the corresponding weights as $W_1 = 0.4$, and $W_2 = 0.6$.
- Two alternate approaches are possible:
  1. Weighted Arithmetic Mean
  2. Normalized Execution Time

- Weighted Arithmetic Mean (WAM):
  - It is computed as the sum of the products of weighting factors and execution times.
  - If W_i denotes the weighting factor of program *i*, we can write

$$\boxed{\text{Weighted Arithmetic Mean} = \sum_{i=1}^{n} (W_i \times XT_i)}$$

Example 1

	CPU A (in secs)	CPU B (in secs)	CPU C (in secs)
Program P1	1	10	25
Program P2	500	250	10
Total Time	501	260	35

- If W_1 = 0.50 and W_2 = 0.50, we get WAM_A = 250.5, WAM_B = 130, WAM_C = 17.5
- If W_1 = 0.90 and W_2 = 0.10, we get WAM_A = 50.9, WAM_B = 34, WAM_C = 23.5
- If W_1 = 0.10 and W_2 = 0.90, we get WAM_A = 450.1, WAM_B = 226, WAM_C = 11.5

(c) Normalized Execution Time

- As an alternative, we can normalize all execution times to a reference machine, and then take the average of the normalized execution times.
  - Followed in the SPEC benchmarks, where a SPARCstation is taken as the reference machine.
- Average normalized execution time can be expressed as either an arithmetic or geometric mean.

$$\boxed{\text{Normalized Arithmetic Mean} = \frac{1}{n}\sum_{i=1}^{n} XTR_i}$$

$$\boxed{\text{Normalized Geometric Mean} = \sqrt[n]{\prod_{i=1}^{n} XTR_i}}$$

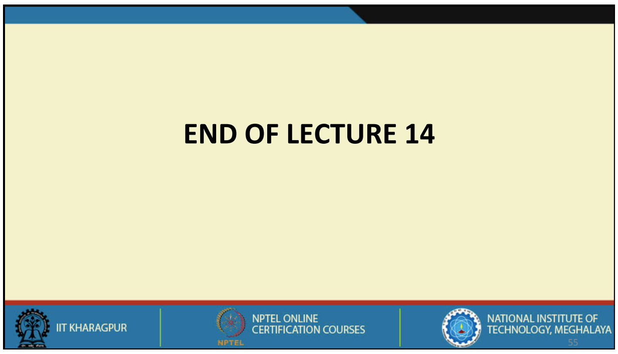
- Here, XTR_i denotes the execution time for the *i*-th program, normalized to the reference machine.

Example 2

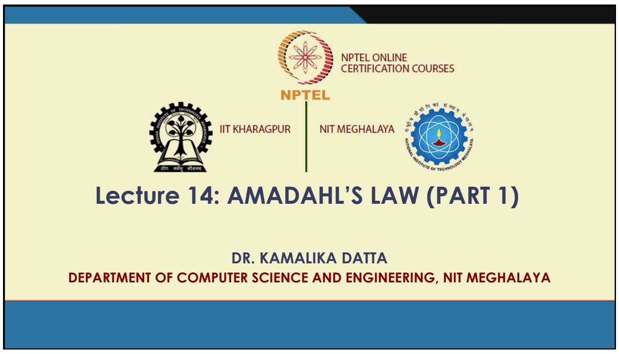
	CPU A (in secs)	CPU B (in secs)	CPU C (in secs)
Program P1	1	10	25
Program P2	500	250	10
Total Time	501	260	35

	Normalized to A			Normalized to B			Normalized to C		
	A	B	C	A	B	C	A	B	C
Program P1	1.0	10.0	25.0	0.1	1.0	2.5	0.04	0.4	1.0
Program P2	1.0	0.5	0.02	2.0	1.0	0.04	50.0	25.0	1.0
Arithmetic mean	1.0	5.25	12.51	1.05	1.0	1.27	25.02	12.7	1.0
Geometric mean	1.0	2.24	0.71	0.45	1.0	0.32	1.41	3.16	1.0

- Summary:
  - In contrast to arithmetic means, geometric means of normalized execution times are consistent no matter which machine is the reference.
  - Hence, the arithmetic mean should not be used to average normalized execution times.
  - One drawback of geometric mean is that they do not predict execution times.
    - Also can encourage hardware and software designers to focus their attention to those benchmarks where performance is easiest to improve rather than the ones that are the slowest.



END OF LECTURE 14



NPTEL ONLINE CERTIFICATION COURSES

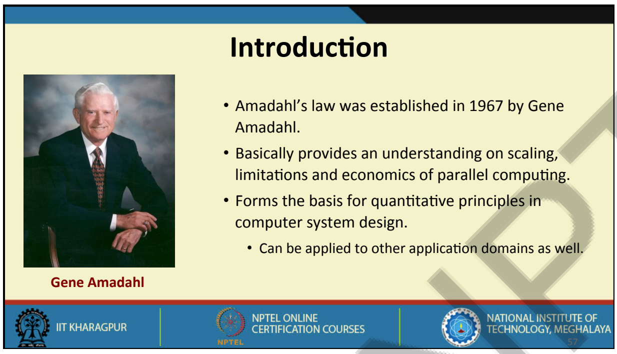
NPTEL

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 14: AMADAHL'S LAW (PART 1)

DR. KAMALIKA DATTA

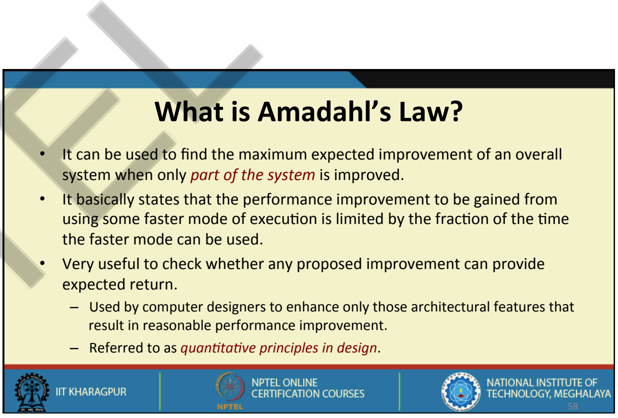
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, NIT MEGHALAYA



Introduction

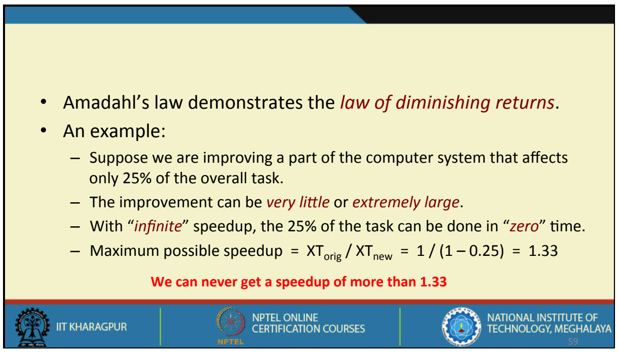
Gene Amadahl

- Amadahl's law was established in 1967 by Gene Amadahl.
- Basically provides an understanding on scaling, limitations and economics of parallel computing.
- Forms the basis for quantitative principles in computer system design.
  - Can be applied to other application domains as well.



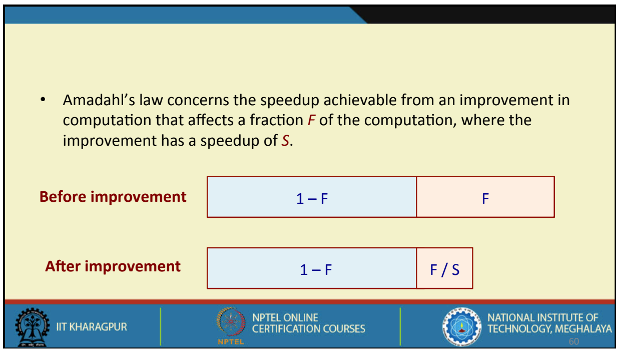
What is Amadahl's Law?

- It can be used to find the maximum expected improvement of an overall system when only *part of the system* is improved.
- It basically states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
- Very useful to check whether any proposed improvement can provide expected return.
  - Used by computer designers to enhance only those architectural features that result in reasonable performance improvement.
  - Referred to as *quantitative principles in design*.

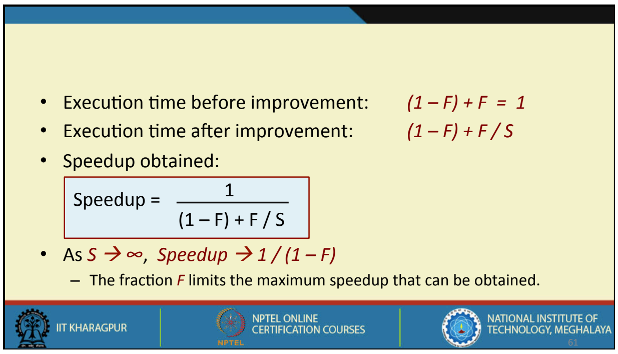


- Amadahl's law demonstrates the *law of diminishing returns*.
- An example:
  - Suppose we are improving a part of the computer system that affects only 25% of the overall task.
  - The improvement can be *very little* or *extremely large*.
  - With "*infinite*" speedup, the 25% of the task can be done in "*zero*" time.
  - Maximum possible speedup = XT_{orig} / XT_{new} = 1 / (1 – 0.25) = 1.33

We can never get a speedup of more than 1.33



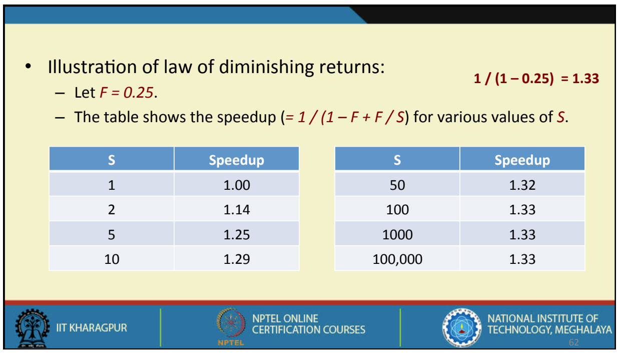
- Amadahl's law concerns the speedup achievable from an improvement in computation that affects a fraction *F* of the computation, where the improvement has a speedup of *S*.



- Execution time before improvement: *(1 – F) + F = 1*
- Execution time after improvement: *(1 – F) + F / S*
- Speedup obtained:

$$\boxed{\text{Speedup} = \frac{1}{(1 - F) + F / S}}$$

- As *S* → ∞, *Speedup* → *1 / (1 – F)*
  - The fraction *F* limits the maximum speedup that can be obtained.

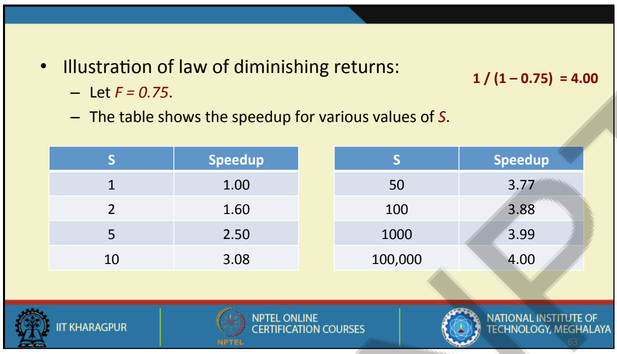


- Illustration of law of diminishing returns:
  - Let *F = 0.25*.
  - The table shows the speedup (= *1 / (1 – F + F / S)*) for various values of *S*.

1 / (1 – 0.25) = 1.33

S	Speedup
1	1.00
2	1.14
5	1.25
10	1.29

S	Speedup
50	1.32
100	1.33
1000	1.33
100,000	1.33

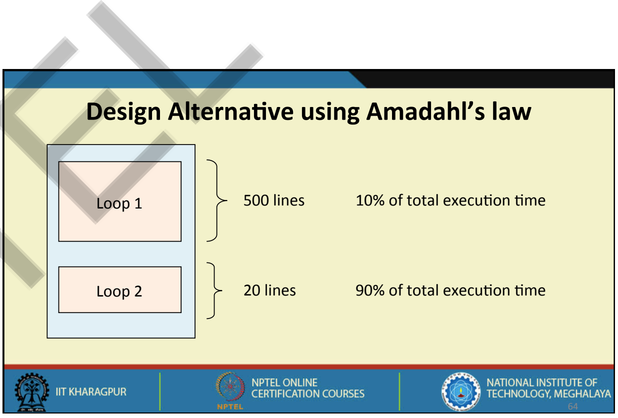


- Illustration of law of diminishing returns:
  - Let *F = 0.75*.
  - The table shows the speedup for various values of *S*.

1 / (1 – 0.75) = 4.00

S	Speedup
1	1.00
2	1.60
5	2.50
10	3.08

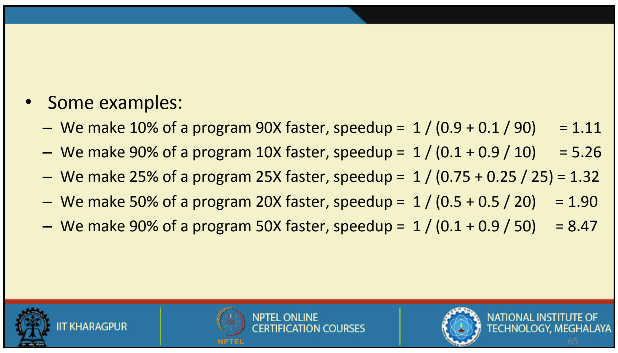
S	Speedup
50	3.77
100	3.88
1000	3.99
100,000	4.00



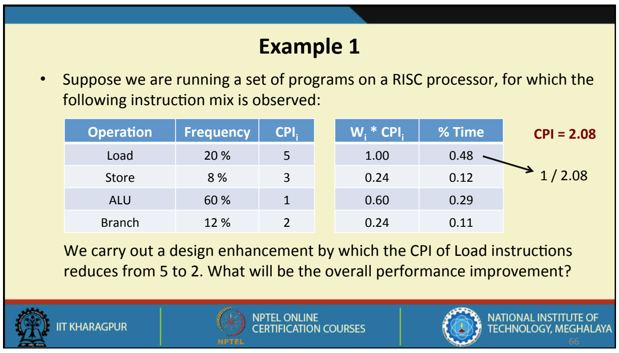
Design Alternative using Amadahl's law

Loop 1 — 500 lines — 10% of total execution time

Loop 2 — 20 lines — 90% of total execution time



- Some examples:
  - We make 10% of a program 90X faster, speedup = 1 / (0.9 + 0.1 / 90) = 1.11
  - We make 90% of a program 10X faster, speedup = 1 / (0.1 + 0.9 / 10) = 5.26
  - We make 25% of a program 25X faster, speedup = 1 / (0.75 + 0.25 / 25) = 1.32
  - We make 50% of a program 20X faster, speedup = 1 / (0.5 + 0.5 / 20) = 1.90
  - We make 90% of a program 50X faster, speedup = 1 / (0.1 + 0.9 / 50) = 8.47



Example 1

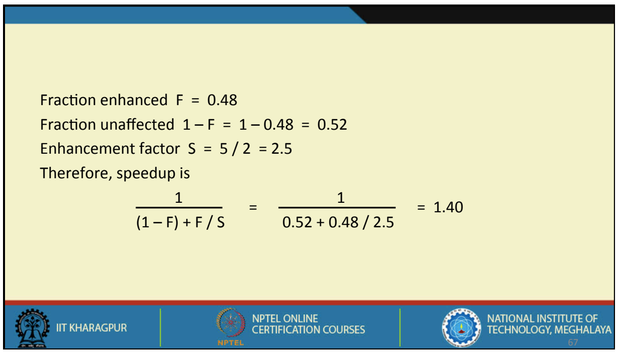
- Suppose we are running a set of programs on a RISC processor, for which the following instruction mix is observed:

Operation	Frequency	CPI_i	W_i * CPI_i	% Time
Load	20 %	5	1.00	0.48
Store	8 %	3	0.24	0.12
ALU	60 %	1	0.60	0.29
Branch	12 %	2	0.24	0.11

CPI = 2.08

1 / 2.08

We carry out a design enhancement by which the CPI of Load instructions reduces from 5 to 2. What will be the overall performance improvement?



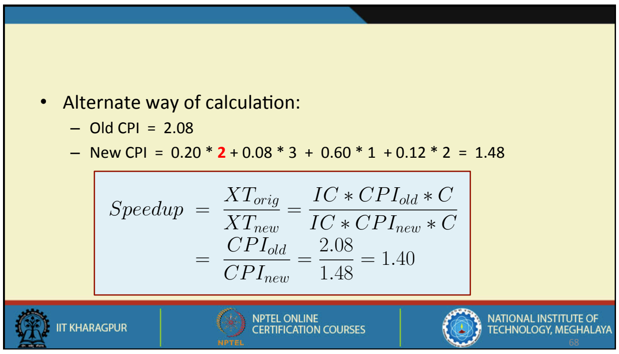
Fraction enhanced F = 0.48

Fraction unaffected 1 – F = 1 – 0.48 = 0.52

Enhancement factor S = 5 / 2 = 2.5

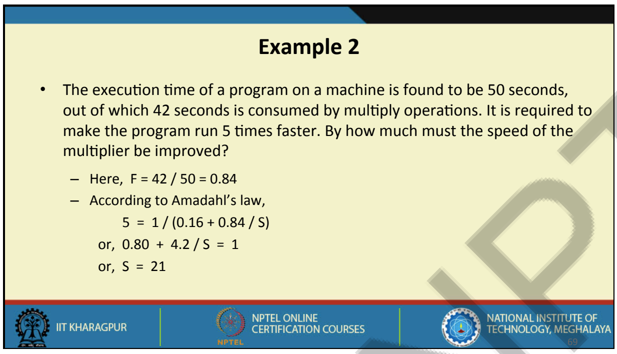
Therefore, speedup is

$$\frac{1}{(1 - F) + F / S} = \frac{1}{0.52 + 0.48 / 2.5} = 1.40$$



- Alternate way of calculation:
  - Old CPI = 2.08
  - New CPI = 0.20 * 2 + 0.08 * 3 + 0.60 * 1 + 0.12 * 2 = 1.48

$$Speedup = \frac{XT_{orig}}{XT_{new}} = \frac{IC * CPI_{old} * C}{IC * CPI_{new} * C}$$
$$= \frac{CPI_{old}}{CPI_{new}} = \frac{2.08}{1.48} = 1.40$$



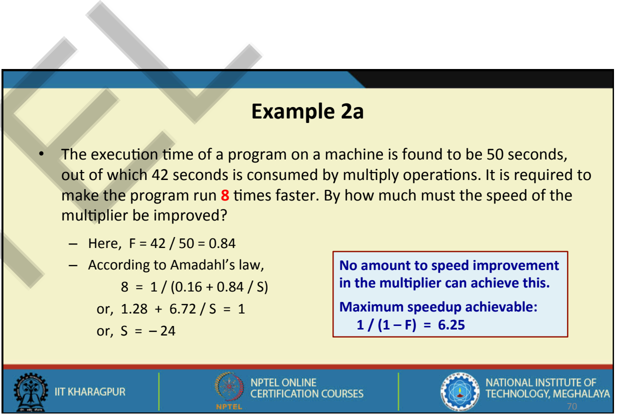
Example 2

- The execution time of a program on a machine is found to be 50 seconds, out of which 42 seconds is consumed by multiply operations. It is required to make the program run 5 times faster. By how much must the speed of the multiplier be improved?
  - Here, F = 42 / 50 = 0.84
  - According to Amadahl's law,

    5 = 1 / (0.16 + 0.84 / S)

    or, 0.80 + 4.2 / S = 1

    or, S = 21



Example 2a

- The execution time of a program on a machine is found to be 50 seconds, out of which 42 seconds is consumed by multiply operations. It is required to make the program run 8 times faster. By how much must the speed of the multiplier be improved?
  - Here, F = 42 / 50 = 0.84
  - According to Amadahl's law,

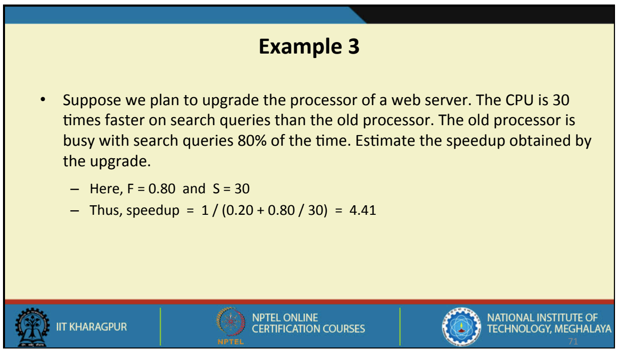
    8 = 1 / (0.16 + 0.84 / S)

    or, 1.28 + 6.72 / S = 1

    or, S = – 24

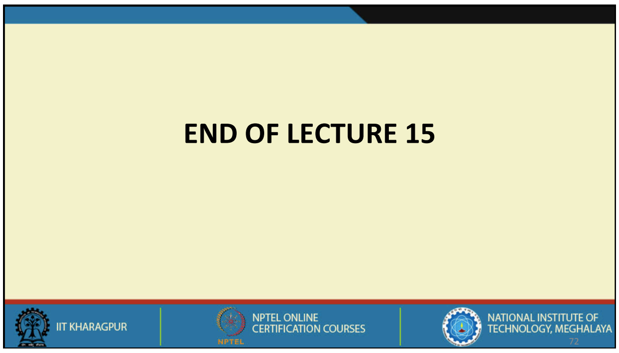
No amount to speed improvement in the multiplier can achieve this.

Maximum speedup achievable:
1 / (1 – F) = 6.25

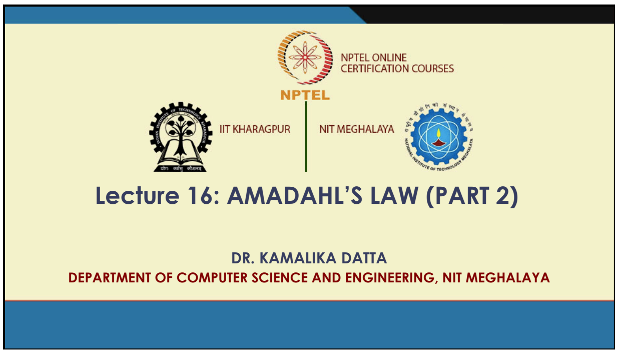


Example 3

- Suppose we plan to upgrade the processor of a web server. The CPU is 30 times faster on search queries than the old processor. The old processor is busy with search queries 80% of the time. Estimate the speedup obtained by the upgrade.
  - Here, F = 0.80 and S = 30
  - Thus, speedup = 1 / (0.20 + 0.80 / 30) = 4.41



END OF LECTURE 15



NPTEL ONLINE CERTIFICATION COURSES

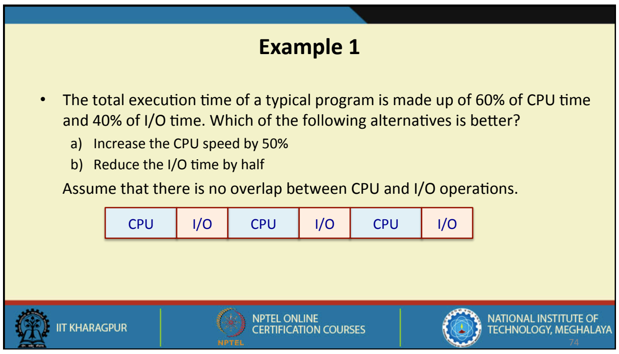
NPTEL

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 16: AMADAHL'S LAW (PART 2)

DR. KAMALIKA DATTA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, NIT MEGHALAYA

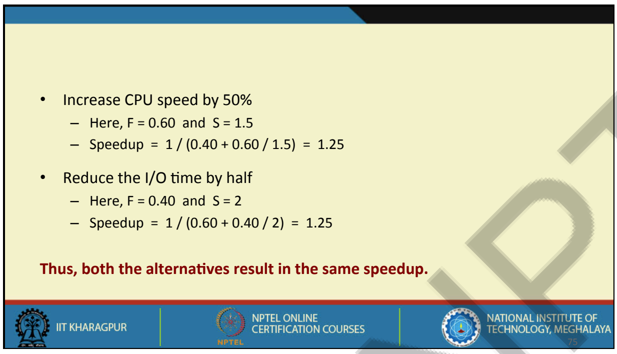


Example 1

- The total execution time of a typical program is made up of 60% of CPU time and 40% of I/O time. Which of the following alternatives is better?
  a) Increase the CPU speed by 50%
  b) Reduce the I/O time by half

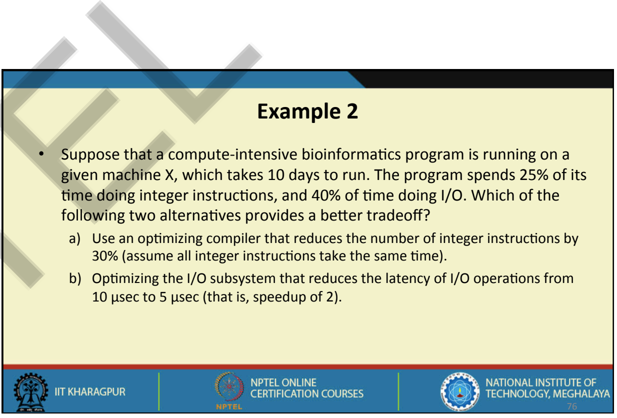
  Assume that there is no overlap between CPU and I/O operations.

CPU	I/O	CPU	I/O	CPU	I/O



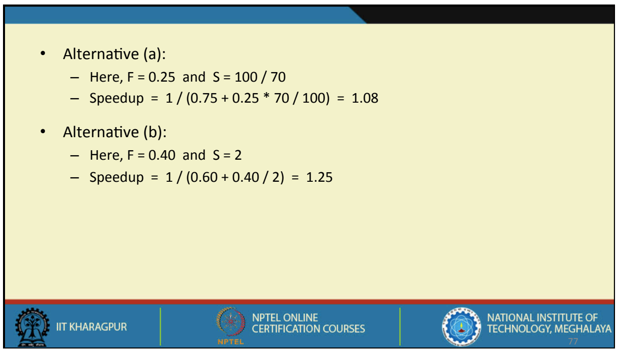
- Increase CPU speed by 50%
  - Here, F = 0.60 and S = 1.5
  - Speedup = 1 / (0.40 + 0.60 / 1.5) = 1.25
- Reduce the I/O time by half
  - Here, F = 0.40 and S = 2
  - Speedup = 1 / (0.60 + 0.40 / 2) = 1.25

Thus, both the alternatives result in the same speedup.

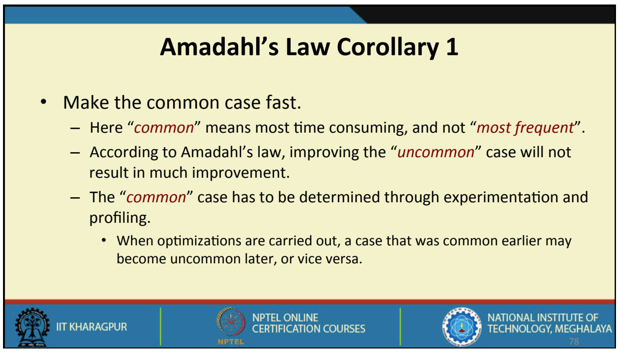


Example 2

- Suppose that a compute-intensive bioinformatics program is running on a given machine X, which takes 10 days to run. The program spends 25% of its time doing integer instructions, and 40% of time doing I/O. Which of the following two alternatives provides a better tradeoff?
  a) Use an optimizing compiler that reduces the number of integer instructions by 30% (assume all integer instructions take the same time).
  b) Optimizing the I/O subsystem that reduces the latency of I/O operations from 10 μsec to 5 μsec (that is, speedup of 2).

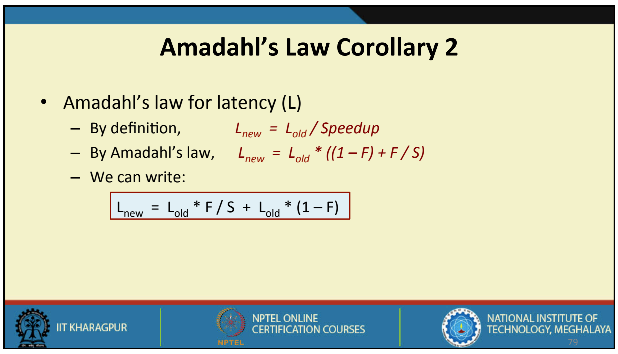


- Alternative (a):
  - Here, F = 0.25 and S = 100 / 70
  - Speedup = 1 / (0.75 + 0.25 * 70 / 100) = 1.08
- Alternative (b):
  - Here, F = 0.40 and S = 2
  - Speedup = 1 / (0.60 + 0.40 / 2) = 1.25



Amadahl's Law Corollary 1

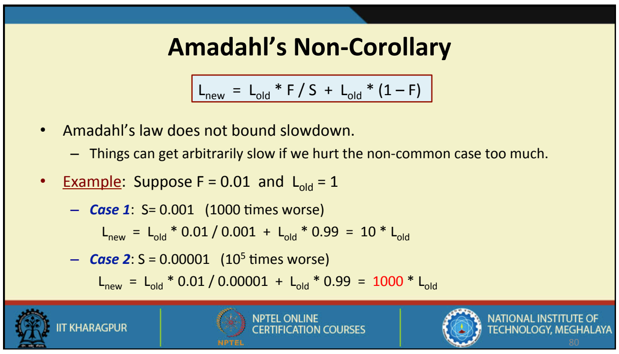
- Make the common case fast.
  - Here "*common*" means most time consuming, and not "*most frequent*".
  - According to Amadahl's law, improving the "*uncommon*" case will not result in much improvement.
  - The "*common*" case has to be determined through experimentation and profiling.
    - When optimizations are carried out, a case that was common earlier may become uncommon later, or vice versa.



Amadahl's Law Corollary 2

- Amadahl's law for latency (L)
  - By definition, $L_{new} = L_{old} / Speedup$
  - By Amadahl's law, $L_{new} = L_{old} * ((1 – F) + F / S)$
  - We can write:

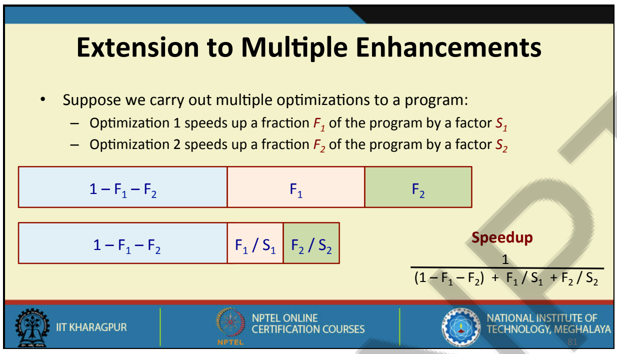
$$\boxed{L_{new} = L_{old} * F / S + L_{old} * (1 – F)}$$



Amadahl's Non-Corollary

$$\boxed{L_{new} = L_{old} * F / S + L_{old} * (1 – F)}$$

- Amadahl's law does not bound slowdown.
  - Things can get arbitrarily slow if we hurt the non-common case too much.
- Example: Suppose F = 0.01 and $L_{old} = 1$
  - *Case 1*: S= 0.001 (1000 times worse)
    $L_{new} = L_{old} * 0.01 / 0.001 + L_{old} * 0.99 = 10 * L_{old}$
  - *Case 2*: S = 0.00001 (10^5 times worse)
    $L_{new} = L_{old} * 0.01 / 0.00001 + L_{old} * 0.99 = 1000 * L_{old}$



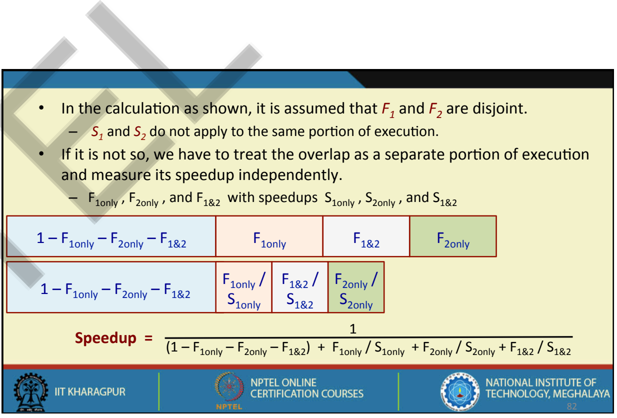
Extension to Multiple Enhancements

- Suppose we carry out multiple optimizations to a program:
  - Optimization 1 speeds up a fraction F_1 of the program by a factor S_1
  - Optimization 2 speeds up a fraction F_2 of the program by a factor S_2

$1 – F_1 – F_2$	F_1	F_2
$1 – F_1 – F_2$	F_1 / S_1	F_2 / S_2

Speedup

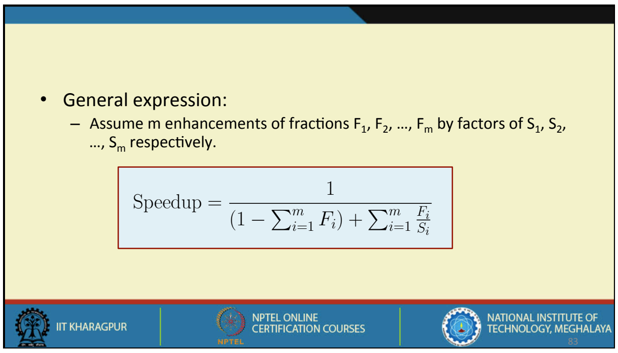
$$\frac{1}{(1 – F_1 – F_2) + F_1 / S_1 + F_2 / S_2}$$



- In the calculation as shown, it is assumed that F_1 and F_2 are disjoint.
  - S_1 and S_2 do not apply to the same portion of execution.
- If it is not so, we have to treat the overlap as a separate portion of execution and measure its speedup independently.
  - F_{1only} , F_{2only} , and $F_{1\&2}$ with speedups S_{1only} , S_{2only} , and $S_{1\&2}$

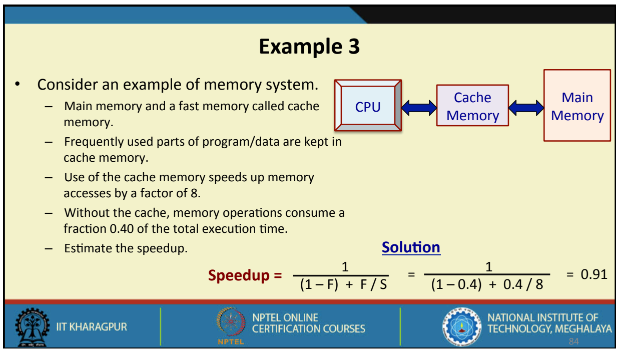
$1 – F_{1only} – F_{2only} – F_{1\&2}$	F_{1only}	$F_{1\&2}$	F_{2only}
$1 – F_{1only} – F_{2only} – F_{1\&2}$	F_{1only} / S_{1only}	$F_{1\&2} / S_{1\&2}$	F_{2only} / S_{2only}

$$\textbf{Speedup} = \frac{1}{(1 – F_{1only} – F_{2only} – F_{1\&2}) + F_{1only} / S_{1only} + F_{2only} / S_{2only} + F_{1\&2} / S_{1\&2}}$$



- General expression:
  - Assume m enhancements of fractions F_1, F_2, …, F_m by factors of S_1, S_2, …, S_m respectively.

$$\boxed{\text{Speedup} = \frac{1}{(1 - \sum_{i=1}^{m} F_i) + \sum_{i=1}^{m} \frac{F_i}{S_i}}}$$



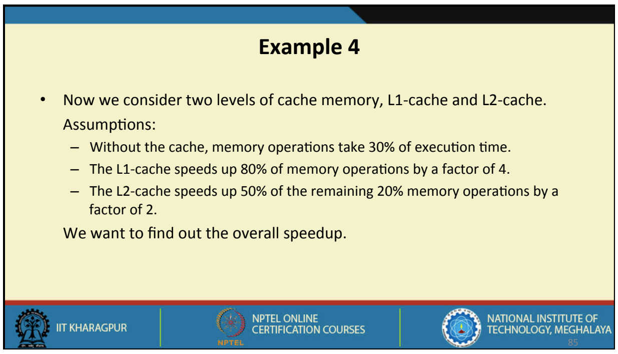
Example 3

- Consider an example of memory system.
  - Main memory and a fast memory called cache memory.
  - Frequently used parts of program/data are kept in cache memory.
  - Use of the cache memory speeds up memory accesses by a factor of 8.
  - Without the cache, memory operations consume a fraction 0.40 of the total execution time.
  - Estimate the speedup.

CPU
Cache Memory
Main Memory

Solution

$$\textbf{Speedup} = \frac{1}{(1 – F) + F / S} = \frac{1}{(1 – 0.4) + 0.4 / 8} = 0.91$$



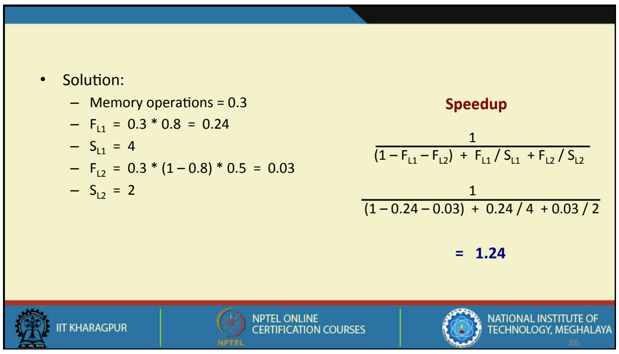
Example 4

- Now we consider two levels of cache memory, L1-cache and L2-cache.
  Assumptions:
  - Without the cache, memory operations take 30% of execution time.
  - The L1-cache speeds up 80% of memory operations by a factor of 4.
  - The L2-cache speeds up 50% of the remaining 20% memory operations by a factor of 2.

  We want to find out the overall speedup.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

85



- Solution:
  - Memory operations = 0.3
  - $F_{L1} = 0.3 * 0.8 = 0.24$
  - $S_{L1} = 4$
  - $F_{L2} = 0.3 * (1 - 0.8) * 0.5 = 0.03$
  - $S_{L2} = 2$

Speedup

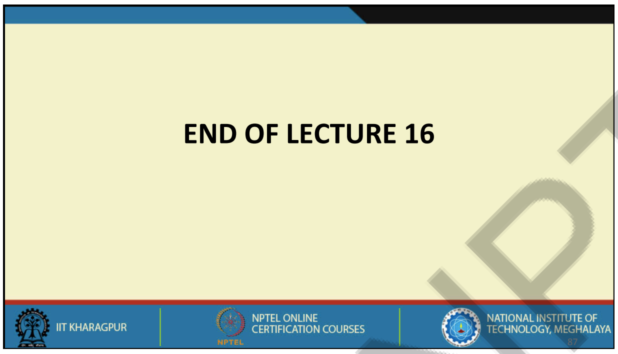
$$\frac{1}{(1 - F_{L1} - F_{L2}) + F_{L1} / S_{L1} + F_{L2} / S_{L2}}$$

$$\frac{1}{(1 - 0.24 - 0.03) + 0.24 / 4 + 0.03 / 2}$$

$$= 1.24$$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

86



END OF LECTURE 16

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

87