Measuring and Reporting Performance

Measuring performance

- An efficient computer design is reducing response time, increasing throughput.

  Execution time or response time* or *elapsed time- it is the latency to complete a task, including disk accesses, memory accesses, input/output activities, operating system overhead everything. Or it is the time between the start and the completion of an event.

  Throughput—it is the total amount of work done in a given time.
- With multiprogramming, the processor works on another program while waiting for I/O and may not necessarily minimize the elapsed time of one program. It is known as *CPU time* means the time the processor is computing, *not* including the time waiting for I/O or running other programs.
- To evaluate a new system the users would simply compare the execution time of their *workloads*—the mixture of programs and operating system commands that users run on a computer.
- In comparing design alternatives, we often want to relate the performance of two different computers, say, X and Y. The phrase "X is faster than Y" is used here which means that the response time or execution time is lower on X than on Y for the given task. In particular, "X is *n* times faster than Y" will mean:

$$\frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

- Since execution time is the reciprocal of performance, the following relationship holds,

$$n = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = \frac{\frac{1}{\text{Performance}_Y}}{\frac{1}{\text{Performance}_X}} = \frac{\text{Performance}_X}{\text{Performance}_Y}$$

Benchmarks

- To measure the performance of a system in real application, a set of benchmark programs are used. The same set of programs can be run on different machines & the execution times compared. Benchmark provides guidance to customers for buying better system based on performance; it is also helpful for the vendors to design systems to meet the benchmark target.
- A benchmark suite is a collection of programs defined in high level language that together attempt to provide test for computer in a particular application or system programming area.
  One of the standardized benchmark application suites has been the SPEC (Standard Performance Evaluation Corporation), whose purpose is to deliver better benchmarks for workstations. Many benchmarks have also been developed for PCs running the Windows operating system.

Desktop Benchmarks

- Desktop benchmarks divide into two broad classes: processor-intensive benchmarks and graphics-intensive benchmarks, although many graphics benchmarks include intensive processor activity.
- SPEC originally created a benchmark set focusing on processor performance (SPEC CPU2006) . The SPEC CPU suite is useful for processor benchmarking for both desktop systems and single-processor servers.
- SPEC CPU2006 consists of a set of 12 integer benchmarks (CINT2006) and 17 floating-point benchmarks (CFP2006)
- The integer benchmarks vary from part of a C compiler to a chess program to a quantum computer simulation. The floating-point benchmarks include structured grid codes for finite element modeling, particle method codes for molecular dynamics, and sparse linear algebra codes for fluid dynamics.

- A server has multiple functions, so are there multiple types of benchmarks. The simplest benchmark is perhaps a processor throughput-oriented benchmark.
- SPEC CPU2000 uses the SPEC CPU benchmarks to construct a simple throughput benchmark where the processing rate of a multiprocessor can be measured by running multiple copies (usually as many as there are processors) of each SPEC CPU benchmark and converting the CPU time into a rate. This leads to a measurement called the SPECrate, and it is a measure of request-level parallelism.
- SPEC offers both a file server benchmark (SPECSFS) and a Web server benchmark (SPECWeb).
- SPECSFS is a benchmark for measuring NFS (Network File System) performance using a script of file server requests; it tests the performance of the I/O system (both disk and network I/O) as well as the processor. SPECSFS is a throughput-oriented benchmark but with important response time requirements.
- SPECWeb is a Web server benchmark that simulates multiple clients requesting both static and dynamic pages from a server, as well as clients posting data to the server.
- SPECjbb measures server performance for Web applications written in Java.
- The most recent SPEC benchmark is SPECvirt_Sc2010, which evaluates end-to-end performance of virtualized datacenter servers, including hardware, the virtual machine layer, and the virtualized guest operating system.
- Transaction-processing (TP) benchmarks measure the ability of a system to handle transactions that consist of database accesses and updates. E.g. Airline reservation systems and bank ATM systems.

Example:
Suppose that the SPECRatio of computer A on a benchmark was 1.25 times higher than computer B;
We know that SPECRatio is a ratio rather than an absolute execution time; the mean must be computed using the *geometric* mean. The execution times on the reference computer drop out.

$$1.25 = \frac{\text{SPECRatio}_A}{\text{SPECRatio}_B} = \frac{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_A}}{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_B}} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{\text{Performance}_A}{\text{Performance}_B}$$

Using the geometric mean ensures two important properties:
1. The geometric mean of the ratios is the same as the ratio of the geometric means.
2. The ratio of the geometric means is equal to the geometric mean of the performance ratios.

$$\frac{\text{Geometric mean}_A}{\text{Geometric mean}_B} = \frac{\sqrt[n]{\prod_{i=1}^{n} \text{SPECRatio A}_i}}{\sqrt[n]{\prod_{i=1}^{n} \text{SPECRatio B}_i}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{SPECRatio A}_i}{\text{SPECRatio B}_i}}$$

$$= \sqrt[n]{\prod_{i=1}^{n} \frac{\dfrac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{A_i}}}{\dfrac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{B_i}}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Execution time}_{B_i}}{\text{Execution time}_{A_i}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Performance}_{A_i}}{\text{Performance}_{B_i}}}$$

$$\text{Geometric mean} = \sqrt[n]{\prod_{i=1}^{n} sample_i}$$

← (Formula of geometric mean)

Reporting Performance Results

- The principle of reporting performance measurements should be *reproducibility*— list everything so that another experimenter would need to duplicate the results.
- A SPEC benchmark report requires an extensive description of the computer and the compiler flags, as well as the publication of both the baseline and optimized results.
- A SPEC report contains the actual performance times, shown both in tabular form and as a graph along with hardware, software and baseline tuning parameter descriptions.
- A TPC benchmark report is even more complete, since it must include results of a benchmarking audit and cost information.
- These reports are excellent sources for finding the real costs of computing systems, since manufacturers compete on high performance and cost-performance.

Quantitative Principles of Computer Design

Quantitative approach of computer design describes how to define, measure, and summarize -- performance, cost, dependability, energy, and power. Following areas can improve the quantitative approach-

1. Take Advantage of Parallelism

It is one of the most important methods for improving performance. The workload of handling requests can then be spread among the processors and disks, resulting in improved throughput. Being able to expand memory and the number of processors and disks is called *scalability*. Spreading of data across many disks for parallel reads and writes enables data-level parallelism.

At the level of an individual processor, taking advantage of parallelism among instructions is critical to achieving high performance. One of the simplest ways to do this is through pipelining. The basic idea behind pipelining is to overlap instruction execution to reduce the total time to complete an instruction sequence. A key insight that allows pipelining to work is that not every instruction depends on its immediate predecessor, so executing the instructions completely or partially in parallel may be possible. Pipelining is the best-known example of instruction-level parallelism.

2. Principle of Locality

The most important program property that we regularly exploit is the *principle of locality*: Programs tend to reuse data and instructions they have used recently. A program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses.

Two different types of locality have been observed. *Temporal locality* states that recently accessed items are likely to be accessed in the near future. *Spatial locality* says that items whose addresses are near one another tend to be referenced close together in time

3. Focus on the Common Case

This is a very simple principle: whenever you have to make a tradeoff **favor the frequent case over the infrequent one**. A common example is related to multiply/divide in a CPU: in most programs the multiplications (integer of float), by far exceed the number of divisions; it is therefore no wonder that many CPUs have hardware support for multiplication (at least for integers) while division is emulated in software. It

is true that in such a situation the division is slow, but if it occurs rarely then the overall performance is improved by optimizing the common case (the multiplication).
This simple principle applies not only to hardware, but to software decisions as well: if you find that some addressing mode, for instance, is heavily used as compared with others than you may try to optimize your design such that this particular addressing mode will run faster, hence increasing the performance of the machine. A fundamental law, called ***Amdahl's law***, can be used to quantify this principle.

Amdahl's Law

- The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's law. Amdahl's law improves the performance by using some faster mode of execution but it is limited by the fraction of the time the faster mode can be used.
- Amdahl's law defines the *speedup* that can be gained by using a particular feature. Speedup is an enhancement to a computer that will improve performance when it is used. Speedup is the ratio that can be evaluated by,

$$\text{Speed up} = \frac{\text{performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

Or

$$\text{Speed up} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

Amdahl's law gives us a quick way to find the speedup from some enhancement, which depends on two factors:-

1.The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement, known as ***fraction enhancement*****, which is always less than or equal to 1.**
Fraction enhanced= time that use enhancement / total execution time

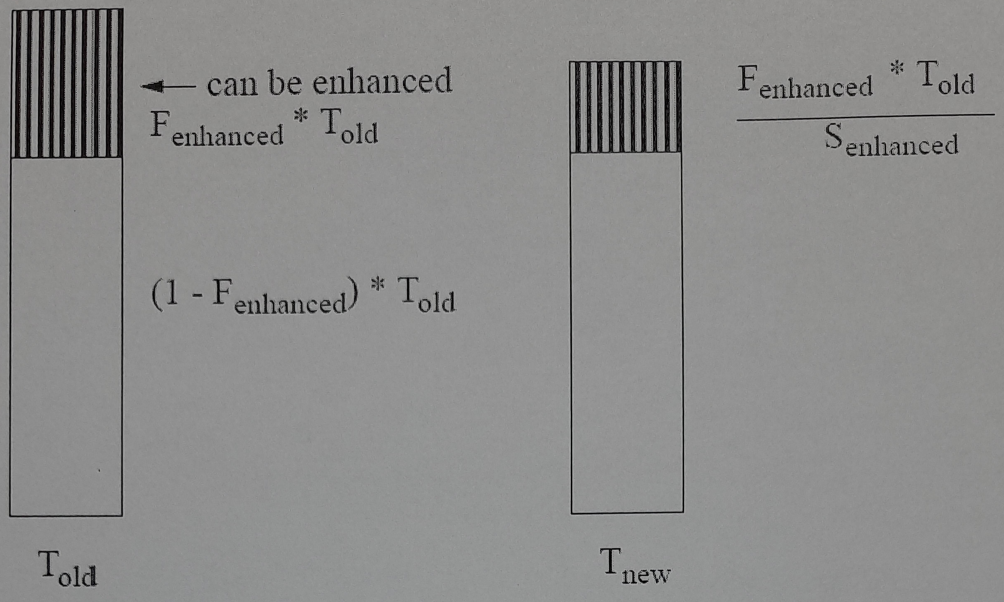
e.g. if 20 seconds of the execution time of a program that takes 60 seconds in total can use an enhancement, the fraction is 20/60. This value, which we will call Fraction enhanced

2.*The improvement gained by the enhanced execution mode, that is, how much faster the task would run if the enhanced mode were used for the entire program, known as speed enhancement*, which **is always greater than 1.**
Speed enhanced = time of original mode / time of enhanced mode

e.g. If the enhanced mode takes, say, 2 seconds for a portion of the program, while it is 5 seconds in the original mode, the improvement is 5/2. We will call this value **Speedup enhanced**

- The execution time using the original computer with the enhanced mode will be the time spent using the unenhanced portion of the computer plus the time spent using the enhancement:

Total execution time = Time spent using the unenhanced portion of the computer
+
Time spent using the enhancement portion.



$$\text{Execution time (new)} = \text{Execution time (old)} \times \left[(1 - \text{Fraction enhanced}) + \frac{\text{Fraction enhanced}}{\text{Speed enhanced}} \right]$$

$$\text{Now, Speed up (overall)} = \frac{\text{Execution time (old)}}{\text{Execution time (new)}}$$

$$= \frac{1}{\left[(1 - \text{Fraction enhanced}) + \frac{\text{Fraction enhanced}}{\text{Speed enhanced}} \right]}$$

Example Suppose that we want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

Answer $\text{Fraction}_{enhanced} = 0.4$; $\text{Speedup}_{enhanced} = 10$; $\text{Speedup}_{overall} = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} \approx 1.56$

Example A common transformation required in graphics processors is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processors designed for graphics. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FPSQR hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for half of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Compare these two design alternatives.

Answer We can compare these two alternatives by comparing the speedups:

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

The Processor Performance Equation

- All computers are constructed using a clock running at a constant rate. These discrete time events are called *ticks*, *clock ticks*, *clock periods*, *clocks*, *cycles*, or *clock cycles*. Computer designers refer to the time of a clock period by its duration (e.g., 1 ns) or by its rate (e.g., 1 GHz).
- CPU time for a program can then be expressed two ways:

CPU time = Total no.of clock cycle for a program × clock cycle time

Or

$$\textbf{CPU time} = \frac{\textbf{CPU clock cycles for a program}}{\textbf{Clock Rate}}$$

- To count the number of clock cycles needed to execute a program, it is required to count the number of instructions executed in a program called ***instruction count*** **(IC).**
- If we know the number of clock cycles and the instruction count, we can calculate the average number of ***clock cycles per instruction*** **(CPI).** [*Instructions per clock (IPC)*, is the inverse of CPI]. CPI is computed as,

$$\textbf{CPI} = \frac{\textbf{Total no.of clock cycle for a program}}{\textbf{Instruction count}}$$

- By transposing the instruction count in the above formula, clock cycles can be defined as IC × CPI. This allows us to use CPI in the CPU execution time formula:

CPU time = Instruction count × no.of clock cycles per instruction × clock cycle time

- Expanding the first formula into the units of measurement shows how the pieces fit together:

$$\frac{\textbf{Instructions}}{\textbf{Program}} \times \frac{\textbf{Clock cycles}}{\textbf{Instruction}} \times \frac{\textbf{Seconds}}{\textbf{Clock cycle}} = \frac{\textbf{Seconds}}{\textbf{Program}} = \textbf{CPU Time}$$

- The number of total processor clock cycles can be calculated as,

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

Where ICi = number of times instruction i is executed in a program
CPIi = average number of clocks per instruction for instruction i.

- Now the CPU time can be expressed as

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

- Overall CPI can be calculated as,

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Instruction count}} \times \text{CPI}_i$$

Other common measurements of performance for processor are- MIPS & MFLOPS

1. MIPS

MIPS (millions of instructions per second) - it is the rate at which instructions are executed expressed as MIPS rate. MIPS rate can be expressed in terms of the clock rate & CPI as follows:

$$\textbf{MIPS rate} = \frac{\text{IC}}{\text{T} \times (10\text{^}6)} = \frac{\text{f}}{\text{CPI} \times (10\text{^}6)}$$

Where IC= instruction count
T= clock cycle duration
f = frequency of processor
CPI= no.of clock cycles per instruction

2. MFLOPS-

MPLOPS (millions of floating-pt operations per second) deals with only floating-pt instructions, mostly in many game and scientific applications. Floating-pt performance is expressed as millions of floating-pt operations per second (MFLOPS), defined as follows:

$$\textbf{MFLOPS rate} = \frac{\text{No.of floating-pt operations executed in a program}}{\text{Execution time} \times (10\text{^}6)}$$

Example-
Consider the execution of a program that result in the execution of 2 million instructions on a 400-MHZ processor. The program consists of four major types of instructions. The instruction mix and the CPI for each instruction type are given below-

Instruction type	CPI	Instructions present of this type (%)
Arithmetic & logical	1	60
Load/store	2	18
Branch	4	12

Ans - Total CPI= (1×0.6) + (2×0.18) + (4×0.12) = 1.44 no.of
f = 400 MHz = 400×(10^6) Hz

$$\text{MIPS rate} = \frac{400\times(10\text{^}6)}{1.44\times(10\text{^}6)} = 277.7$$

OR

IC = 1/ CPI = 1/1.44 = 0.6944 no.of

$$T = 1/f = \frac{1}{400\times(10\text{^}6)}$$

$$\text{MIPS rate} = \frac{0.6944}{\frac{1}{400 \times (10\text{^}6)} \times (10\text{^}6)} = 0.6944 \times 400 = 277.7$$

Example 1.1

Effect of system enhancements on response time, throughput: The following system enhancements are considered: a) Faster CPU

b) Separate processors for different tasks (as in an airline reservation system or in a credit card processing system) do these enhancements improve response-time, throughput or both?

Answer: A faster CPU decreases the response time and, in the mean time, increases the throughput

a) Both the response-time and throughput are increased.

b) Several tasks can be processed at the same time, but no one gets done faster; hence only the throughput is improved.

Comparing Performance

Suppose we have to compare two machines A and B. The phrase *A is n% faster than* B means:

$$\frac{\text{Execution time of B}}{\text{Execution time of A}} = 1 + \frac{n}{100}$$

Because performance is reciprocal to execution time, the above formula can be written as:

$$\frac{\text{Performance A}}{\text{Performance B}} = 1 + \frac{n}{100}$$

Example 1.2 COMPARISON OF EXECUTION TIMES:

If machine A runs a program in 5 seconds, and machine B runs the same program in 6 seconds, how can the execution times be compared?

Answer:

Machine A is faster than machine B by n% can be written as:

$$\frac{\text{Execution\_time\_B}}{\text{Execution\_time\_A}} = 1 + \frac{n}{100}$$

$$n = \frac{\text{Execution\_time\_B} - \text{Execution\_time\_A}}{\text{Execution\_time\_A}} * 100$$

$$n = \frac{6-5}{6} \times 100 = 16.7\%$$