**Chapter 2: Performance Issues**

Table of Contents

[2.1 Designing for Performance 2](#_Toc48981792)

[Microprocessor Speed 2](#_Toc48981793)

[Performance Balance 3](#_Toc48981794)

[Improvements in Chip Organization and Architecture 5](#_Toc48981795)

[2.2 Multicore, MIC and GPGPUs 7](#_Toc48981796)

[2.3 Ahmdahl’s Law and Little’s Law 8](#_Toc48981797)

[Ahmdahl’s Law 8](#_Toc48981798)

[Little’s Law 9](#_Toc48981799)

[2.4 Basic Measures of Computer Performance 10](#_Toc48981800)

[Clock Speed 10](#_Toc48981801)

[Instruction Execution Rate 11](#_Toc48981802)

[2.5 Calculating the Mean 13](#_Toc48981803)

[2.6 Benchmarks and SPEC 16](#_Toc48981804)

[Benchmark Principles 16](#_Toc48981805)

[SPEC Benchmarks 16](#_Toc48981806)

## 2.1 Designing for Performance

Every year, computers get cheaper but have greater performance and capacity. Processors have become so cheap that they can simply be thrown away, while applications have reached astounding levels of complexity and power. The funny part is that the basic building blocks of today’s computers are the same as the ones of the IAS computer. The techniques for squeezing the maximum performance out of the materials used have just become increasingly sophisticated.

### Microprocessor Speed

The evolution of processors continues to meet Moore’s law, which leads to chip makers releasing a new generation of chips every three years, with four times as many transistors. In memory chips, the capacity of DRAM quadruples every three years, while in microprocessors, new circuits with reduced separation distances improves performance four or five-fold every three years. This raw speed however, does not achieve its full potential without constantly being given work in the form of instructions. Anything that slows down the flow of incoming work will hinder the true capabilities of these chips. Processor designers must therefore come up with elaborate techniques to increase this flow. Such techniques include:

* Pipelining: Instructions are executed in multiple stages. This method allows the processor to work on multiple instructions at the same time, putting each instruction at a different stage, similar to how assembly line manufacturing works. For example, while one instruction is being executed, the processor is decoding the next instruction and fetching the one after that and so on.
* Branch Prediction: The processor looks ahead into the instruction code fetched from memory to predict which groups of instructions are likely to be executed next. If it is able to guess correctly, it can prefetch those instructions and buffer them so that there is no delay when their execution begins. More sophisticated examples may prefetch multiple branches instead of just one.
* Superscalar Execution: This is the ability to issue more than one instruction in every clock cycle, thus allowing multiple parallel pipelines.
* Data Flow Analysis: The processor analyses which instructions are dependent on each other’s results, thus creating an optimized schedule of instructions. Instructions are scheduled to be executed when ready instead of the original program order, which prevents unnecessary delay.
* Speculative Execution: Using branch prediction and data flow analysis, some processors execute instructions that are likely to be needed ahead of their actual appearance in the program execution, storing results in temporary locations.

### Performance Balance

While processors have improved at astounding speeds, other important components have failed to keep up. This results in the need to look for performance balance, an adjustment to the organization and architecture of computers to compensate for the mismatch in capabilities of the various components.

The problems created by such mismatches is particularly critical at the interface between the processor and main memory. Processors are extremely fast compared to the speed at which data can be transferred between the two components. This pathway is responsible for carrying a constant flow of data and instructions, so any bottlenecks here results in an idle processor. A few architectural changes that can be made to handle this problem are:

* Using a wider data bus, which increases the number of bits that can be retrieved at a time from memory
* Change DRAM interface to include cache, thus buffering more data
* Include complex and efficient cache structures between the processor and main memory to reduce the frequency of memory access
* Increase the bandwidth of the interconnect by using faster buses

Another major problem is the slow speed of I/O devices. Although modern I/O devices have intensive demands, processors are capable of handling those demands. The problem occurs in transmitting data between processors and I/O devices. Improvement strategies here include caching and buffering schemes, the use of faster and more elaborate interconnection structures and dedicated microprocessors for I/O devices.

The main factor to consider is the balance between demands from different components. We must consider the facts that the rate at which performance is changing in various technological areas is wildly different from one element to the next and that new applications and new peripheral devices are constantly changing the nature of the demand on the system in terms of typical instruction profile and data access patterns.

### Improvements in Chip Organization and Architecture

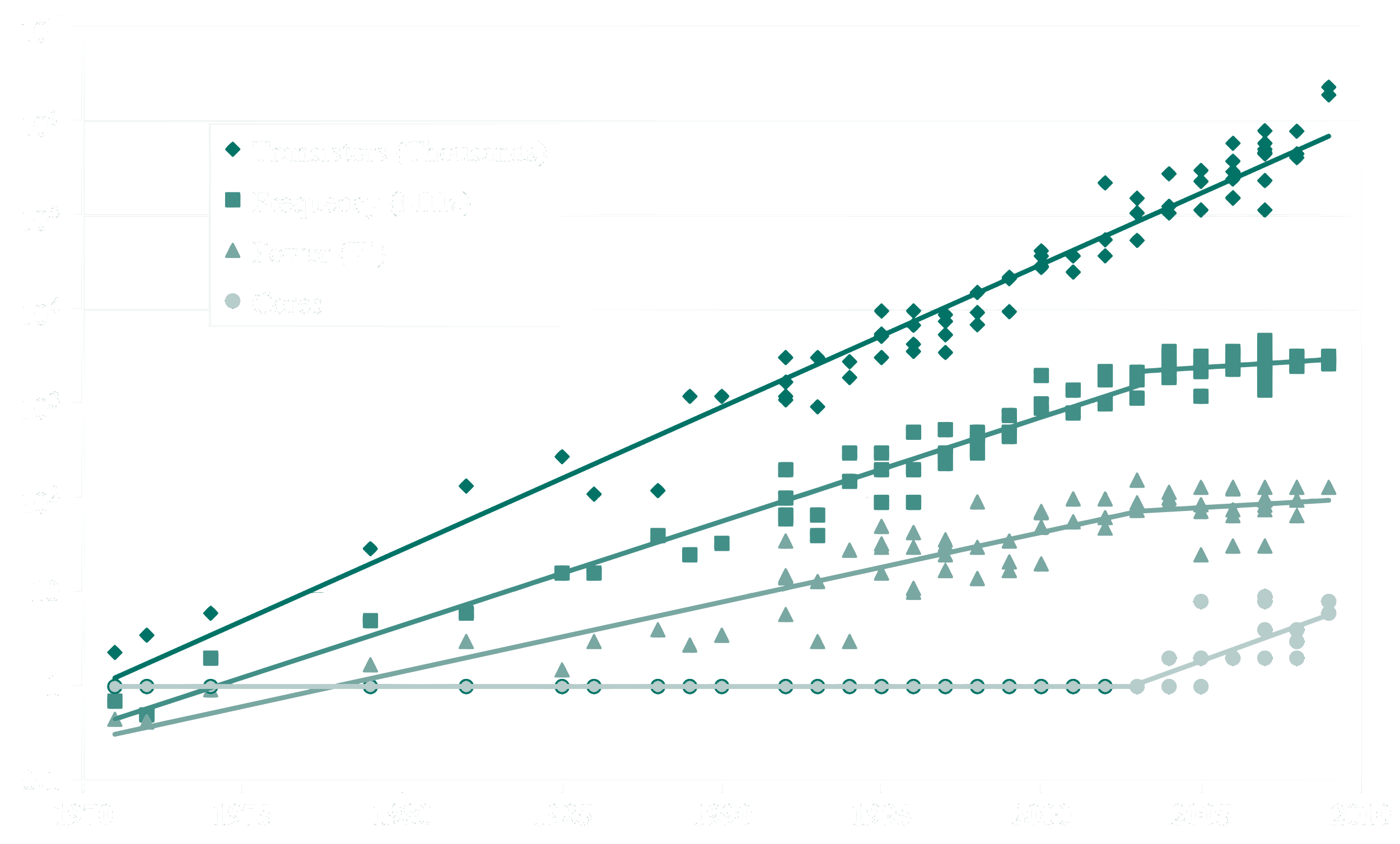
The search for balance does not prevent the need to increase processor speeds. There are three approaches to this:

* Increasing hardware speed by packing gates that are smaller more tightly onto the processor chip. The decreased distance between gates decreases propagation times for signals, making the processor faster. Increasing the clock rate also helps since it means individual operations are executed more rapidly.
* Increasing size and speed of caches between the processor and main memory. Dedicating some space in the processor itself to use as cache (this is now 50%) can drop cache access times significantly.
* Making changes to the organization and architecture of processors to improve the effective speed of instruction execution.

Increasing clock speed and logic density leads to some problems too:

* Heat generated from power is difficult to dispose of due to the high density and speed of chips.
* The speed at which electrons flow through chips is restricted by the resistance and capacitance of the metal wires connecting them, called the RC product. As components get smaller, the wires get thinner and closer together, which increases the RC product which leads to increased delay, called the RC delay.
* Memory access speeds (latency) and transfer speeds (throughput) are slower than processor speeds.

The main strategies being used to increase performance right now are increasing cache capacity, since higher chip density means more cache memory can be put on the chip itself, and making instruction execution logic more complex, to allow parallel execution of instructions. Simply increasing clock rate is no longer an option, since power consumption is becoming too high. The image below shows this trend:



Instead, manufacturers are now relying on making use of the increasing number of transistors, as well as developing multicore chips.

## 2.2 Multicore, MIC and GPGPUs

The concept of a multicore chip is to place multiple processors on the same chip, along with a shared cache. This provides the potential to increase performance without increasing clock rate. Larger caches are also used since power consumption of memory logic is much lower than that of processor logic. There are now levels of caches on the chip, with the first and perhaps second levels being dedicated to individual processors, and those above being shared. Manufacturers are now in the process of making a huge leap forward, to more than 50 cores per chip, which has been given the name many integrated core (MIC).

The multicore and MIC strategies involve a collection of general-purpose processors on a single chip. Another design option is to add graphics processing units (GPUs) onto the same chip, using specialised cores to handle video processing and other tasks. The GPU is designed to perform parallel operations on graphics data. Since GPUs can perform parallel operations on multiple sets of data, they are being used on applications that require repetitive computations. This has blurred the line between a GPU and a CPU, which has given rise to the term General Purpose Computing on GPUs (GPGPUs).

## 2.3 Ahmdahl’s Law and Little’s Law

We will now look at two laws that provide insight into the performance of parallel and multicore systems.

### Ahmdahl’s Law

Advances in technology and changes in design are constantly being made to improve system performance. Use of parallel processors, memory cache hierarchy, speeding up memory access time and I/O transfer rate are all examples. However, it is important to note that speeding up one aspect of technology or design does not necessarily result in a corresponding improvement in overall performance. This limitation is captured through Ahmdahl’s Law.

Ahmdahl’s Law deals with the potential speedup of a program when using multiple processors as opposed to a single processor. It takes a program that has a total execution time , a fraction that is inifinitely parallelizable and a fraction that is sequential. Thus, if processors were used, the change in execution time would only come from the fraction that is capable of running in parallel.

From this equation, we observe two things. First, when is small, the use of parallel processors has little effect. Second, as approaches infinity, we reach an upper bound of , meaning there are diminishing returns for using more processors.

Thus, to properly exploit the power of parallel processing, the software that runs on machines with multiple cores must be adapted to a highly parallel execution environment.

Ahmdahl’s Law can be generalized to evaluate any design or technical improvement to a computer system. For any enhancement to a feature,

For a feature that is used during a fraction of the execution time, and enhancement of that feature results in a speedup for that feature, the overall speedup of the system is

### Little’s Law

We can apply Little’s Law to any system that is statistically in a steady state and in which there is no leakage. Say we have a steady state system in which items arrive at an average rate of items per unit time and stay in the system for an average of units of time. There is an average of units in the system at any one time, so

Using queuing theory terminology, Little’s Law applies to a queuing system. If there is a server providing some service to items, it will serve an item immediately if it is idle. If the server is not idle, incoming items will join a queue. There may be a single queue for one or more servers, or multiple queues, one for each server. This theory can be applied to a processor providing services to processes, a transmission line providing transmission services to packets of data or an I/O device providing read or write services to I/O requests.

## 2.4 Basic Measures of Computer Performance

While evaluating processor hardware and setting new system requirements, performance is one key factor to consider, along with cost, size, security, reliability and power consumption. However, performance is difficult to compare among different processors, even those from the same family. Raw speed is far less important than how a processor actually performs when executing a given application, and that depends on a lot of other factors like the instruction set, implementation language, efficiency of the compiler and skill of programming the application. We need to take all of this into consideration when benchmarking processors.

First, we will look at raw speed, specifically, the clock speed and instruction execution rate.

### Clock Speed

Every operation performed by a processor is governed by a system clock. Typically, each operation begins with the pulse of the clock, which means the speed of the processor is dictated by the pulse frequency, measured in cycles per second, or hertz (Hz). The pulses are generated by a quartz crystal, which generates a constant sine wave while power is applied. This is converted into a digital voltage pulse stream, which in turn goes to the processor circuitry. The rate of pulses is called the clock rate or clock speed. Each pulse is called a clock cycle, or clock tick. The time between pulses is the clock time.

The clock rate is not chosen arbitrarily. It must be appropriate for the physical layout of the processor. Actions in the processor require signals to be sent from on element to another, and when they are sent, it takes some time for things to settle down so that the logic level can accurately be determined. Depending on the physical layout, some signals may change more rapidly than others, so operations must be synchronized and paced to make sure the proper electrical signals are available for each operation.

The execution of a single instruction has multiple steps, like fetching the instruction, decoding it, loading and storing data, etc. This means each instruction will take multiple clock cycles to complete. Additionally, if pipelining is being used, multiple instructions are executing simultaneously. Thus, a simple comparison of clock speeds will not give us enough information to compare processor performance.

### Instruction Execution Rate

A processor is driven by a clock with a constant frequency , or a constant time cycle where . If a given program, or part of program has a total of instructions, and the th instruction , requires cycles to complete, then the average cycles per instruction, , is given by

Thus, the time needed to execute the program can be given by

We can refine this formula by recognizing that each instruction execution actually has two parts, work done by the processor and a word being transferred to or from memory, and that the memory cycle time is different from the processor cycle time, and may be greater. If an instruction takes instruction cycles to be decoded and executed, memory references, and the ratio of memory cycle time to processor cycle time is , then

The five performance factors, , , , and , are influenced by four system attributes, the design of the instruction set (instruction set architecture), compiler technology (how effectively the compiler produces an efficient machine language program from a high-level language program), processor implementation and cache and memory hierarchy.

A common measure for performance of a processor is the rate at which instructions are executed. Since this number is very high, it is expressed in millions as millions of instructions per second (MIPS).

Another common performance measure deals with just floating-point instructions, and is expressed in millions of floating-point operations per second (MFLOPS).

## 2.5 Calculating the Mean

A single number, like execution time or memory consumed, is often used to characterized performance and compare systems, albeit in a very simplified manner. The use of benchmarks to compare systems involves using mean values of data points related to execution. There are a few alternatives to finding this mean value, the three most common being the arithmetic mean, the geometric mean and the harmonic mean.

The arithmetic mean (AM) is appropriate if the sum of all the measurements is a meaningful and interesting value, such as comparing execution time performance for several systems. We could take a program and run it several times with different inputs on a particular system, and find the arithmetic mean of all the runs. This would give us a good measure of the programs performance on that system, and a good number to use to compare between systems. The AM for a time-based variable such as execution time, has the property of being directly proportional to the total time, so if total time doubles, AM doubles.

For some cases, the execution rate may be viewed as more useful. This could be the instruction execution rate or the program execution rate. Say we have a benchmark program with operations, for which we measure the execution times for different systems. Thus, for the th program, the execution rate .

The arithmetic mean execution rate is proportional to the sum of the inverse execution times. This is simply put, a weird thing to figure out. It would have been much nicer if the arithmetic mean had simple been inversely proportional to the sum of the execution times. The harmonic mean however, gives us exactly this:

It is easier to get a feel of how AM and HM behave rather than how geometric mean (GM) behaves. Firstly, with respect to changes in values, GM gives equal importance to all values. A change in 10% in the largest value in a data set will have a considerable effect in AM, while the same change in the smallest value will seem to have no effect. GM on the other hand, will show the same change for both cases.

One observation with GM is that the GM of a ratio is equal to the ratio of the GMs, i.e.

There is one drawback to GM when used with execution times. There is a possibility that, as values increase, the GM may increase or decrease, as opposed to AM which is guaranteed to increase. This is non-intuitive.

We also have something called a functional mean (FM). For a continuous monotonic function, the functional mean for positive real numbers is:

For , the FM is the AM. For , the FM is the GM and for , the FM is the HM.

## 2.6 Benchmarks and SPEC

### Benchmark Principles

Measures like MIPS or MFLOPS are inadequate to evaluate the performance of processors. There are differences in instruction sets (RISC vs CISC) and different types of programs. This has led to systems being tested on benchmark programs, so the same set of programs can be run on different machines and the execution times compared. The characteristics of a benchmark program are:

* Written in high-level language, making it portable across different machines
* Representative of a particular domain, like systems, numerical or commercial
* Easily measurable
* Wide distribution of tasks

### SPEC Benchmarks

We now have standard benchmarks suites for industrial and academic and research communities, that use a collection of benchmark programs that attempt to provide a representation of a system in a particular field. The best-known collection of benchmark suites is from the Standard Performance Evaluation Corporation (SPEC), with the best one being the SPEC CPU2006, which concentrates on measuring performance for applications that spend most of their time with computation rather than I/O. The CPU2006 suite is based on actual applications rather than artificial or synthetic cases, which makes results much more reliable. It consists of 12 integer benchmarks and 17 floating point benchmarks, written in C, C++ and Fortran, and consisting of more than 3 million lines of code.

SPEC documentation uses the following terms regularly:

* Benchmark: Simple, a program
* System Under Test
* Reference Machine: The system used by SPEC to establish a baseline performance for all benchmarks. The benchmark results on the system under test are compared to those of the reference machine.
* Base Metric: Strict guidelines that all compilers must follow during testing
* Peak Metric: These are used to optimize the system performance by changing compilation settings.
* Speed Metric: A measurement of the time it takes to execute a compiled benchmark. This gives us a way to compare the ability of a computer to complete single tasks.
* Rate Metric: Used to measure the rate of task accomplishment.

Calculations done during the assessment of a system is a three-step process. We will look at the 12 programs that are integer benchmarks here, though the process is the same for the 17 floating-point benchmarks.

* First, each program is compiler and run three times, to get the median value of the runtime. This is done to take into account the variations in execution time due to factors outside the program.
* Next, each of the 12 results is normalized by calculating the runtime ratio of the reference runtime to the system runtime.
* Finally, the geometric mean of the 12 runtime ratios is calculated.

For systems with processors, during each run of a program, copies of the program are run simultaneously, and the rate metric is calculated as

Take this problem for example:

|  |  |  |  |
| --- | --- | --- | --- |
| **Benchmark** | **Processor** | | |
|  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

If we are asked to find the arithmetic mean for the speed metrics for the systems and , normalized to system , then we would first need to calculate the speed metric for each system for each benchmark. For , these would just be . For , the first would be , the second and so on. For , we would need to do the same. Then we would need to find the arithmetic mean of our results. Thus, for , the result is , for it is and so on. All mathematical problems related to the equations given above are essential to this course.