Basic Concepts: Performance Equation and Computational Complexity

The ultimate and only reliable metric for measuring processor performance is **execution time**. A computer system typically executes multiple processes (multiprogramming and multi-tasking) during a certain amount of time, using a mechanism called *time-sharing*: The Operating System (OS) schedules the execution of multiple processes, which swap between being running on the Central Processing Unit (CPU) and being waiting in an OS queue. Many of the processes are devoted to general tasks like serving interrupts generated from the Input/Output devices, or executing maintenance tasks. If several users are connected to the computer, their running programs fight for getting a CPU execution slot, and the **elapsed time** for all the processes may increase. You should always check that the execution of your program takes more than 99% of the total CPU resources; for that purpose, you can use the **top** or the **perf** commands.

In order to explain the performance of our program, it is necessary to identify the quantitative value for all the factors that determine the elapsed execution time. The part of the elapsed time where the program is not running (waiting time) is explained by the contention with other execution process. We are interested on the **CPU time**, the time where the program is running using the CPU. The next equation is used to "explain" the CPU time of a program in terms of several factors. This equation is commonly found in books regarding processor architecture and performance engineering.

$$\text{CPU Time} = \text{instructions} / (\text{IPC} \times \text{clock frequency})$$

A program can execute faster by executing **less machine instructions**. Machine instructions (for brevity we say instructions) are a small subset of primitive operations and the ones that the computer really understands and executes. Instructions perform simple operations (copy, addition, comparison …) on data operands of basic types (32-bit integers, 64-bit real numbers in floating-point representation, 8-bit characters …). You can reduce the total number of instructions executed by using a better algorithm (computation strategy), or by codifying the algorithm more efficiently using a high-level programming language. The compiler is responsible for translating between high-level language sentences and low-level machine instructions. You can provide some information to help the compiler improve its work. You can also activate or deactivate compiler optimization strategies by using compiler flags on the invocation of the compilation process (like **-O3** or **-fno-inline**). The machine instruction code can be inspected in the form of assembly language, but it is specific for a certain processor architecture.

Machine instructions are executed at the rate driven by a clock signal. The **clock frequency** determines how many clock ticks happen per second (3 GHz means 3 thousand million clock ticks per second). Execution time can be measured in clock ticks, instead of seconds. Obviously, the higher the clock frequency, the better.

Finally, the IPC measures the **ratio of machine instructions executed per clock cycle**. The higher the IPC, the better. A higher IPC indicates that the processing hardware is more efficient when executing the machine instructions. In order to explain the variations in the IPC it is necessary to understand the basics of the architecture of a modern processor. First, processors are able to execute several machine instructions simultaneously, and an IPC higher than one is sometimes possible. Second, processors contain internal cache memory that is much faster than the external DRAM memory, therefore, if most of the program's data is accessed from the cache memory, the IPC will be higher than in the case where most of the data is accessed from main memory.

There can be different ways to solve the same problem using a computer. The sequence of steps to solve a problem is called an algorithm. The **computational** or **temporal complexity** of an algorithm is defined as the number of simple operations or steps performed by the algorithm as a function of the problem size. Different algorithms used to solve the same problem may have different complexities: for example, Bubble Sort takes n^2 steps to sort n numbers, while Merge Sort takes $n \times \log_2 n$ steps. An operation is a step in the algorithm repeated many times, and depends on the application. The programmer or engineer decides what to consider an operation, looking for a simple and precise definition.

The **memory complexity** of the algorithm is the total amount of memory that the program needs to execute. It is determined by the size of the data structures that must be "*alive*" at some point of the execution of the program.

Estimating the amount of instructions required to execute a high-level operation is not straightforward: we will later analyze this question in more depth. However, we can measure this number of instructions using profiling tools.

How can we compare the performance for different problem sizes? Since one expects that the CPU time should be proportional to the amount of work performed, we can measure performance as the **ratio of operations per time**. Therefore, we can normalize execution time and express performance as **operations per second**. The analogy is the performance measurement of a car expressed in Km/hour. A sorting algorithm with complexity n^2 will perform 4x more work when doubling n, so we expect that the execution time will also be multiplied by 4x.

Performance expressed as work per second can be "explained" as the multiplication of three simpler factors: (1) the **codification efficiency** (average number of operations that are codified per machine instruction, or per thousands of machine instructions); (2) the **microarchitecture throughput** (or IPC, which is the average number of machine instructions executed per clock cycle);

and (3) the H/W speed, measured as clock cycles per second (or clock frequency). One advantage of this new equation is that both performance and all the three factors follow the rule that "*higher is better*", which is what most people intuitively assume when analyzing a graphic for the first seconds.

$$\frac{\text{Operations}}{\text{Second}} = \frac{\text{Operations}}{\text{Instruction}} \; \frac{\text{Instructions}}{\text{Clock Cycle}} \; \frac{\text{Clock Cycles}}{\text{Second}}$$

OpRate IPC Clock Freq.

Codification or coding efficiency depends on how well the compiler translates the statements expressed in a high-level language, like C, or Java, or FORTRAN, or Matlab, into low-level machine instructions. Changing the compiler or the processor means that a different codification may be obtained, probably requiring more or less machine instructions.

Microarchitecture throughput (rate of instructions executed per cycle or IPC) depends on the internal organization (or *microarchitecture*) of the processor. Most processors today can execute multiple instructions simultaneously, giving an average IPC higher than one. Many events may prevent achieving the maximum efficiency of the processor, like the occurrence of data dependences between instructions, especially those affecting long-latency memory load operations. Those performance hazards are related to the actual program instructions, so at the end of the day the achieved IPC depends both on the program characteristics and on the processor microarchitecture.

Finally, the *clock frequency* mostly depends on the technology used to implement a processor and the cleverness of the engineers designing the silicon gates and internal connections. High clock frequencies imply high-energy consumptions (power consumption has approximately a cubic dependence on clock frequency) so that using lower clock frequencies is a way to save energy at the expense of compute performance.

See also: Algorithm (https://en.wikipedia.org/wiki/Algorithm)
Analysis of Algorithms (https://en.wikipedia.org/wiki/Analysis_of_algorithms)

See also: Time-Sharing (https://en.wikipedia.org/wiki/Time-sharing),
Elapsed Time (https://en.wikipedia.org/wiki/Elapsed_real_time)
CPU Time (https://en.wikipedia.org/wiki/CPU_time)