Chapter 1

Performance Measures

Reading: The corresponding chapter in the 2nd edition is Chapter 2, in the 3rd edition it is Chapter 4 and in the 4th edition it is Chapter 1.

When selecting a computer, there are different criteria a user might want to consider

- Whether the application one wants to run is available for that computer
- Energy consumption
- Cost
- Often, the most important factor is: **performance** (good resolution, smooth execution, not too slow)

The goal of this chapter is to define more precisely what we mean by "performance" and how we go about measuring it.

1.1 Measuring performance

What do we mean if we say that one computer has a performance that is better than another computer? The answer is not obvious. That defining performance is not unambiguous is illustrated by the following example.

Example 1.1.1

The following table lists the characteristics of three different airplanes.

	# of passengers	Range	Speed
Boeing 747	470	4150	610
Concorde	146	4000	1360
Douglas DC-8	132	8720	544

(a) If you are one of the passengers, who wants to get to her/his destination as fast as possible, your best pick would be the airplane with the highest speed, i.e., the Concorde.

(b) If you are the president of the airline company, and you want to maximize the number of passengers that you can transport in a day between Paris and New York ($3,000$ miles route), your best pick would be the Boeing 747: although it is about two times slower than the Concorde, so the Concorde could make two round trips in the time the Boeing makes a single roundtrip, the Boeing can transport more than three times as many people, per flight, so it could still transport the most passengers per day.

(c) If you want to buy a private airplane and your primary concern is to be able to fly direct from Paris to San Diego ($5,500$ miles), without stopping, than the Douglas DC-8 would be your best pick. The range of the other two airplanes is not sufficient to fly directly from Paris to San Diego.

As the example illustrates, the optimal choice can vary depending on what criterion one uses to define "optimal". Similarly, there are different ways to define "performance" for computers. For example, once could define it as:

- The "response time", also called "execution time" or "latency": the time it takes from start to completion of a task (in the airplane analogy: fly one passenger from Paris to New York; and, thus, choose the Concorde)
- The "throughput": the total amount of work completed in a given time interval (in the airplane analogy: the total number of passengers that can be transported in a given time interval, for example, one day; and, this, choose the Boeing 747)

In this class, we will focus on execution time to measure performance.

1.2 Execution time

1.2.1 Definition

Execution time is the time between starting and completing a task. We will denote execution time as ET.

1.2.2 Performance

Performance is denoted by PF and is related to the execution time ET as

$$PF = \frac{1}{ET} \ .$$

Indeed: to maximize the performance, we would want to minimize the execution time, for a given task.

Example 1.2.1

If the performance of machine A is ten times better than machine B, what is the relation between their execution times?
Answer: Since $PF_A = 10 \cdot PF_B$, we have $\frac{1}{ET_A} = 10 \cdot \frac{1}{ET_B}$ and thus $ET_B = 10 \cdot ET_A$. So, A being ten times better than B indeed implies that B needs ten times more time than A to execute a given task, or B is ten times slower than A.

1.2.3 CPU Time

Measuring the time that elapses from the start to the completion of a task would include the time needed to execute the task on the CPU, the time to access the hard drive when needed, the time waiting for I/O (input/output) from the user, which could be extremely long, the time required for OS (operation system) overhead (often, several tasks are being executed simultaneously and the OS determines when the CPU is allowed to work on which task; for that, the OS needs to spend some time, which is called OS overhead), the time spent by other programs in the CPU, while the current task is not yet completed, etc.

In the end, what we are really interested in is the "**CPU time**": the time the CPU spends computing for a given task (not including the time spent waiting for I/O, running other programs, accessing the hard drive, etc.). The CPU time can be further divided in:

$$\text{CPU time} = \begin{cases} \textbf{user CPU time} : \text{time the CPU spends on running the actual program} \\ \text{system CPU time} : \text{time the CPU spends on OS overhead for the program} \end{cases}$$

The CPU spends time on OS overhead for the program when the OS performs tasks on behalf of the program, like, e.g., reloading registers, etc. Usually, we will use the **user CPU time** as execution time (ET) and call the corresponding performance measure the "**CPU performance**", or "**performance**" in short. We will talk about "**System performance**" when we want to refer explicitly to the time spent, by the CPU, both on the actual program and the OS overhead on behalf of the program, i.e., the entire CPU time.

1.3 Measuring Execution Time

1.3.1 Clock cycles

All components in a computer, including the CPU, are operating at the pulse of a "clock". The clock in a computer

- runs at a constant rate,
- determines WHEN events (computing a sum, saving information, charging a capacitor) can take place in the CPU and other computer components: during one "clock cycle", any transistor can only make one switch.

We will denote the number of clock cycles that are needed to execute a program as $\#CC$ and the duration of one clock cycle T, the "**clock period**", respectively. The clock frequency f is defined as $\frac{1}{T}$. Note that the units of T are *sec* or *msec*, and the units of f are *Hz*, *MHz* or *GHz*.

The execution time for a program can now be obtained as follows:

$$\text{ET} = \text{user CPU Time} = \#CC \cdot T = \frac{\#CC}{f} \ .$$

This expression suggests how a hardware designer could potentially increase the performance of a machine. Indeed, the execution time could either by decreased by decreasing $\#CC$, the number of clock cycles required to execute the program, or by increasing f, the clock frequency. However, there is a tradeoff between decreasing $\#CC$ and increasing f, hardwarewise.
Indeed, on one hand, reducing $\#CC$ requires more complex hardware. For example, $(A+B+C)$ requires 2 additions (and thus 2 clock cycles) if we only have a 2-input adder available. It can be done in 1 instruction (and one clock cycle) using a 3-input adder, which requires more complex hardware. On the other hand, to allow more complex hardware, the clock frequency f might need to decrease. Indeed, since the CPU is running at the pulse of the clock, the hardware designer must ensure that all electric signals have physically propagated correctly by the end of each clock cycle. Since more complex hardware results in larger propagation delays, hardware designers are required to lower the clock frequency f in more complex systems.

1.3.2 Clock cycles per instruction

The clock period T (or, clock frequency f) depends directly on the hardware specs. However, the number of clock cycles $\#CC$, depends both on the hardware and the software, i.e, the specific sequence of low-level instructions we need to execute to complete a given task. Let us define the CPI as the average number of clock cycles it takes to execute one instruction and assume that our program consists of I instructions. Then, the execution time (ET) for our program can be expressed as:

$$\text{ET} = \text{user CPU Time} = CPI \cdot I \cdot T = \frac{CPI \cdot I}{f} \ .$$

Note that the number of clock cycles may be different for different types of instructions. If this is the case, one can first calculate the overall, average CPI amongst all types of instructions and then use the previous equation to compute the execution time. The next example illustrates this point.

Example 1.3.1

`ADD` takes 1 clock cycle and `MULT` takes 3 clock cycles. If a program consists of 20 `ADD` and 10 `MULT` instructions, and f is 1GHz, what is the average CPI and the execution time?

Answer: Let's denote the CPI for an `ADD` instruction as $CPI1$, and the CPI for a `MULT` instruction as $CPI2$. Then we have $CPI1 = 1$ (clock cycle / instruction), $CPI2 = 3$ (clock cycles /instruction). The average CPI can be calculated as:

$$\begin{aligned}\text{Average CPI} &= \frac{\text{Total number of clock cycles}}{\text{Total number of instructions}} \\ &= \frac{\text{CPI1} \cdot (\#\text{ of } \texttt{ADD}) + \text{CPI2} \cdot (\#\text{ of } \texttt{MULT})}{(\#\text{ of } \texttt{ADD}) + (\#\text{ of } \texttt{MULT})} \\ &= \frac{20 \cdot 1 + 10 \cdot 3}{20 + 10} = \frac{5}{3} = 1.67 \text{ clock cycles / instruction .}\end{aligned}$$

Now, we can calculate ET as:

$$\text{ET} = \frac{\text{Average CPI} \cdot \text{Total number of instructions}}{f} = 50 \text{ nsec .}$$

Expressing ET in terms of the CPI, I (the number of the instructions) and f allows us to understand the software-hardware tradeoffs that are involved in increasing the CPU performance. To reduce the execution time (the user CPU Time), one can decrease T, the CPI or I. We already discussed the hardware trade offs this implies. Similarly, there is trade off between decreasing the CPI and decreasing the number of the instructions, from a software perspective. Indeed, if a program consists only of simple instructions, the average CPI will be small (hardware only requires few clock cycles to execute simple instructions), but the number of instructions required to execute a given task will be large since simple instructions only provide simple functionality. Inversely, programming with complex instructions will decrease the number of the instructions, but result in a high CPI. The bottom line is that intelligent concurrent hardware/software design is very important to build the best performing machines.

1.4 Other Performance Measures

Besides execution time (usually defined as user CPU time), several other performance measures are commonly used like, for example, MIPS and MFLOPS. MIPS is defined as follows:

$$\begin{aligned}\text{MIPS} &= \text{Millions of Instructions Per Second} \\ &= \frac{I}{ET \cdot 10^6} \\ &= \frac{I}{\#CC \cdot T \cdot 10^6} \\ &= \frac{I}{CPI \cdot I \cdot T \cdot 10^6} \\ &= \frac{1}{CPI \cdot T \cdot 10^6} = \frac{f}{CPI \cdot 10^6} \text{ .}\end{aligned}$$

MFLOPS is Millions of FLoating-point Operations Per Second. Floating point operations are often used to measure performance because they're usually slower than other instructions

and therefore the performance bottleneck. In the next example, we take a closer look at MIPS as a performance measure. Although it is widely used, it is not always the best way to measure performance.

Example 1.4.1

Assume one has written a program in C and compiled it with two different compilers. Each compiler generated the following numbers of instructions of each of three instruction types, A, B and C, for this program (expressed in millions, M):

	A	B	C
Compiler 1	5M	1M	1M
Compiler 2	10M	1M	1M

Let f be 1GHz. An instruction of type A, B, and C takes 1, 2 and 3 clock cycles, respectively.

(a) ET

$$\text{ET1} = \frac{\#CC_1}{f} = \frac{CPI_A I_{1A} + CPI_B I_{1B} + CPI_C I_{1C}}{f} = \frac{10^6(5\cdot 1 + 1\cdot 2 + 1\cdot 3)}{10^9} = 10 \text{ msec} ,$$
$$\text{ET2} = \frac{\#CC_2}{f} = \frac{CPI_A I_{2A} + CPI_B I_{2B} + CPI_C I_{2C}}{f} = \frac{10^6(10\cdot 1 + 1\cdot 2 + 1\cdot 3)}{10^9} = 15 \text{ msec} .$$

(b) MIPS

$$\text{MIPS1} = \frac{I_{1A} + I_{1B} + I_{1C}}{\text{ET1}\cdot 10^6} = \frac{7\cdot 10^6}{10\cdot 10^{-3}\cdot 10^6} = 700 ,$$
$$\text{MIPS2} = \frac{I_{2A} + I_{2B} + I_{2C}}{\text{ET2}\cdot 10^6} = \frac{12\cdot 10^6}{15\cdot 10^{-3}\cdot 10^6} = 800 .$$

Based on execution time, we conclude that Compiler 1 generates the faster program, i.e., it is better than Compiler 2. However, the code from Compiler 2 has a higher MIPS rating, which would make it better than Compiler 1.

The above example demonstrates that MIPS is not always an appropriate performance measure. The main reason for this is that MIPS does not take into account the complexity of the instructions, i.e, how much each instruction achieves towards the completion of a given task. Indeed, executing many simple instructions per second can lead to a high MIPS rating, however, executing fewer but more complex instructions per second could still complete the task in less time, total. Therefore, just using MIPS to compare machines with different instruction set architectures is not advised. Moreover, MIPS can even vary inversely with the actual execution time between programs on the same machine, as illustrated in the previous example. Overall, *ET* (user CPU Time) is usually a better performance measure than MIPS.

1.5 Amdahl's Law

Gene Amdahl (1922) realized that focusing on improving the slowest component of an architecture significantly is not enough to improve the overall performance proportionally.

Amdahl's Law: Do not expect that the improvement of one component of a system will

improve the overall system proportionally. If one component is improved by the factor S, then the overall ET_{Improved} is given by

$$ET_{\text{Improved}} = \left(\frac{ET_{\text{affected}}}{S} + ET_{\text{unaffected}}\right) ,$$

where the system is partitioned in a part that is affected by the improvement and a part that is not affected by the improvement.

Example 1.5.1

A program consists of two types of instructions, A and B. Assume the program spends 80 nsec running A-type instructions and 20 nsec on B-type instructions.

(a) How long does it take to execute the program?
Answer: 80 + 20 = 100 nsec

(b) Assume we improve the system such that only 16 nsec are spent on instructions of type A (500 % improvement). How long does it take to execute the program? How much has the system performance improved?
Answer: 16 + 20 = 36 nsec. Thus, we have a $\frac{100}{36} \cdot 100 = 278\%$ improvement.

(c) If A-type instructions only take 1.6 nsec to be executed (5000 % improvement), how much does the system performance improve?
Answer: 1.6 + 20 = 21.6 nsec. Thus, we have a $\frac{100}{21.6} \cdot 100 = 463\%$ improvement.

Note that even though we have dramatically improved the performance of one type of instructions, the overall improvement is not as significant (500% vs. 278% and 5000% vs. 463%).