Chapter 6
Measuring Performance

One accurate measurement is worth a thousand expert opinions.
– G. Hopper

He uses statistics as a drunken man uses lampposts: for support rather than illumination.
– A. Lang

Abstract  Previously we developed a simple processor data-path; our motivation was simply to demonstrate the fundamentals of how such a processor might work. We can measure the performance of such a processor according to a huge number of criteria. For example, we might consider how many instructions it executes in a given time period, how much power it uses, how large the manufactured circuit is, and so on. In a more real-life context, we might tune our design to give the best performance according to one of these criteria; in this chapter we investigate a number of ways to measure performance (focusing on the speed at which a processor executes programs).

6.1 Measuring Performance

Before we start to examine better processor designs we need to be able to evaluate their relative performance characteristics, i.e., understand what “better” means. The problem is that performance can be a difficult concept to define exactly. It is often more instructive to think about the overall quality of a design rather than performance. Quality can combine lots of different features as dictated by different application or market areas; a cost function will determine how quality is evaluated by combining many metrics. For example, one might worry about how quickly a processor executes a given program, or how much power is used during execution. In most designs, such metrics compete for resources in the sense that no one can be satisfied without being detrimental to another; some form of trade-off or compro-
mise must be reached, one can think of this as maximising the cost function in some way.

For a general circuit, perhaps the most simple metric one can consider is how quickly it operates. This can be further decomposed into two standard measures.

**Definition 32.** The **latency** of a circuit is the total time elapsed before a given input is operated on to produce an output. The **bandwidth** or **throughput** of a circuit is the rate at which new inputs can be offered (or outputs produced).

The difference is somewhat subtle but we will see why both are important when we look at improving performance: for now, simply consider latency. From the basic definition, one can try to capture what is meant by performance.

**Definition 33.** The **performance** of design $X$ is inversely proportional to latency

$$ \text{PERFORMANCE}(X) = \frac{1}{\text{LATENCY}(X)}. $$

From this, we can define what is meant by the statement “design $X$ is $n$ times faster than design $Y$”. That is, if $n$ is the speed-up offered by design $X$ over design $Y$, then

$$ \text{SPEED-UP}(X \text{ over } Y) = \frac{\text{LATENCY}(Y)}{\text{LATENCY}(X)} = \frac{\text{PERFORMANCE}(X)}{\text{PERFORMANCE}(Y)} = n. $$

In the context of a processor rather than a general circuit, one typically uses **execution time** in place of latency; that is, the time taken to execute a given program. However, when placed in our definition this presents even more problems: what do we mean by execution time, how do we measure time and what exactly are we executing?

### 6.1.1 Estimating Execution Time

There are plenty of bad ways to evaluate the performance of a processor that are none-the-less still used in marketing literature. A prime example is the **MIPS** metric, a different MIPS from the MIPS32 ISA! Essentially the MIPS metric measures how many instructions a processor can execute each second: the assertion is that the more instructions executed each second, the higher the performance. One might therefore try to estimate the execution time of program $X$ on processor $Y$ using

$$ \text{EXECUTION-TIME}(X,Y) = \frac{\text{INSTRUCTION-COUNT}(X)}{\text{MIPS-RATING}(Y)}. $$
This is an intuitive, easy to understand measure with which one can compare the performance of processors. In fact, in the 1970s the performance of computers was quoted relative to a baseline computer; the VAX 11/780 was said to represent 1 VAX MIPS, the VAX executed one million instructions per-second. The problem is, this method of comparison is quite flawed. In particular, it should be clear that a RISC processor will typically take many instruction to execute a given program since each instruction performs a relatively simple operation; a CISC processor will typically take less instructions since each instruction performs more complex operations. Comparing one with the other is nonsense: we need a performance metric that includes the idea of amount of useful work done. That is, a RISC processor will almost always have a higher MIPS rating than a CISC alternative; regardless of the design quality. In addition, the MIPS rating is tied to the clock speed of the physical processor. In theory at least, it would be nice to evaluate the quality of a design independently from the clock speed.

The Cycles Per-Instruction (CPI) metric hopes to solve this problem to some extent. It measures the average number of clock cycles needed per-instruction and varies according to the processor architecture and the program being executed. The instruction mix for a processor tells us what ratio of different instruction types are used in the average program. For example, we might note that 40% of instructions perform arithmetic, 40% perform memory access and 20% perform branches. The process of discovering the instruction mix for a processor is called workload characterisation; we typically use a range of benchmark programs to produce an overall picture of how often different instructions are used.

Given the instruction mix and the CPI for each instruction type, we can estimate the overall CPI for the processor. The CPI for each instruction is easily identified; typically they are quoted as part of the processor manual. Let $CPI_i$ denote the CPI for instruction type $i$ and $F_i$ denote how often instruction type $i$ is executed. The overall CPI is then calculated as

$$CPI = \sum_{i=0}^{n-1} F_i \cdot CPI_i.$$  

Consider an example instruction mix:

<table>
<thead>
<tr>
<th>Operation</th>
<th>$F_i$</th>
<th>$CPI_i$</th>
<th>$F_i \cdot CPI_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>40%</td>
<td>1</td>
<td>0.4</td>
</tr>
<tr>
<td>Memory</td>
<td>40%</td>
<td>4</td>
<td>1.6</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>0.4</td>
</tr>
<tr>
<td>Total</td>
<td>100%</td>
<td></td>
<td>2.4</td>
</tr>
</tbody>
</table>

Using the overall CPI, we can then estimate how long a given program will take to execute. Firstly we can estimate the number of cycles the program will take to execute; this requires we know the number of instructions executed by the program and the overall CPI for the processor as computed above. Then, for program $X$ on processor $Y$ we have
\[ \text{CYCLES}(X,Y) = \text{INSTRUCTION-COUNT}(X) \cdot \text{CPI}(Y). \]

Finally we can estimate the execution time by dividing the number of cycles by the clock rate:

\[
\text{EXECUTION-TIME}(X,Y) = \frac{\text{CYCLES}(X)}{\text{CLOCK-RATE}(Y)}
\]

\[
= \frac{\text{INSTRUCTION-COUNT}(X) \cdot \text{CPI}(Y)}{\text{CLOCK-RATE}(Y)}
\]

### 6.1.2 Measuring Execution Time

Rather than estimating execution time of a given program, one might simply measure how long it takes to execute. Typically we use the term **wall-clock time** to mean the elapsed time for execution. However, this may include components other than the workload under examination such as time to load the program, time spent waiting for access to resources such as storage devices and so on. Depending on what we want to get from the exercise, we might include or exclude these components. Time is then one of:

**Wall-clock Time**  Also known as execution time or response time, this is the total time required to execute a program, including all the potential overheads.

**Processor Time**  Unlike wall-clock time which includes overheads, processor time is the total computational time spent during execution i.e., without overheads. That is, processor time is the amount of time the program spends having instructions executed by the processor.

**System Time**  The amount of time spent executing operating system or library code on behalf of the program under examination. This might include calls to functions that load data from a storage device for example.

**User Time**  In contrast with system time, user time measures how much time is spent executing the program itself.

Most UNIX systems include the `time` command which is able to report wall-clock, system and user times for the execution of a given program. The resolution (or accuracy) of the results is not ideal, but it offers an easy first attempt at measuring how long a program takes to execute. On a UNIX system for example, we might use the `wc` program to count the number of words in a local dictionary named `/usr/share/dict/words`; with the `time` command we can measure how long this takes using the following:

```
bash$ time wc /usr/share/dict/words
483523 483523 4992010 /usr/share/dict/words
real 0m0.714s
user 0m0.656s
sys 0m0.011s
```
We can see that the total time taken was 0.714 seconds, of which only about 0.656 was spent executing the actual \texttt{wc} program.

Listing 6.1 and Listing 6.2 demonstrate two common methods of measuring execution time of a program fragment with a higher degree of accuracy; in both cases the continuation dots represent the fragment being timed. Listing 6.1 uses the C \texttt{clock} function which returns an approximation of how much time has elapsed since the beginning of execution. It has roughly a microsecond accuracy. By recording the value before the fragment and afterwards, the total time taken is the difference between the two. It is common to divide by \texttt{CLOCKS_PER_SEC}, a processor-specific constant that allows conversion into a number of seconds. Deviating for a moment from our use of MIPS32 as an example, Listing 6.2 is specific to Intel
processors. It uses some inline assembly language to access the `rdtsc` instruction that returns the **Time Stamp Counter (TSC)** value. The TSC is a register internal to the processor, an example of a machine-specific register. Every time a processor cycle is completed, the register is incremented; the program reading the TSC can thus measure time, very precisely, in number of processor cycles.

### 6.1.3 Benchmark Programs

Although we now know how to measure or estimate the execution time for a given program on a given processor, another problem remains. Specifically, it is not clear *which* program we should be measuring (and what input data to provide) so we get a meaningful result. Clearly these choices will have a major effect on our analysis. For example, a program that performs mostly memory accesses will probably perform differently than a program that performs a lot of computation; input data that is hard to process or causes worst-case behaviour in the constituent algorithms will provoke worse performance than more friendly input. As such, to get an idea of processor performance that is independent of the program being executed, we need to make choices that match typical usage. That is, since we cannot measure the execution time of all possible programs using all possible inputs, we need to find a representative sample of programs (and input data) that can be easily extrapolated to how the processor will really be used.

A **benchmark** is a program or set of programs selected and used to act as the required sample. Benchmarks allow one to perform comparisons between processor designs by simply measuring and comparing execution time, and to extrapolate to other programs by performing workload characterisation and deriving typical instruction mixes. A benchmark should, in general, be representative of the type of applications run on the processor, not be overly dependent on specifics of the processor (unless one wants to test a specific feature), and be reproducible in the sense that we compare and verify results. Benchmarks can usually be placed in one of three main classes:

**Application** A benchmark application is a single program, specifically selected, or even specifically written, to test particular aspects of performance. Examples include 3DMark series of benchmarks which are used to test the performance of 3D graphics rendering in modern graphics cards. Using a single application enables one to focus on specific items of interest but the results are often not easy to extrapolate because of this.

**Suite** A benchmark suite is typically a collection of benchmark applications. Where a single application tests only one aspect of performance, the suite captures many aspects by constructing some form of average over all the component applications. This gives less specific results but makes it easier to see trends that apply in general. Examples include the SPEC benchmarks, original versions of which included commonly used UNIX programs such as `gzip` and `ghostview`. 
Kernel  A benchmark kernel is a very small fragment of program; it might not do anything useful alone, but represents the typical computational core of the programs we are interested in. Kernels are usually easy to program and analyse but do not usually test overall system performance. Examples include the Livermore Loops [41] a set of Fortran, and later C, kernels that compute common arithmetic functions such as matrix multiplication and vector dot product. They focus mainly on program fragments that loop over arrays of data reasoning that this is the main computational core of scientific applications.

6.1.4 Measuring Improvement

Our initial definition of the speed-up given by design $X$ over design $Y$ is somewhat naive. To be more accurate, we need to consider Amdahl’s Law which, roughly speaking, states that the performance improvement of using some faster mode of execution is limited by the fraction of the time the faster mode can be used. In short, Amdahl’s Law is just a slightly technical way of stating the well-known law of diminishing returns. That is, if an improvement implemented in design $X$ is not used very often the benefit in performance will be small: the less it is used, the smaller the improvement will be.

Imagine we improve some hardware design (perhaps some part of the ALU) $H$ within in a basic processor design $Y$ to produce a new processor design $X$. Also, suppose the design for $H$ in processor $X$ speeds-up the original design in processor $Y$ by some factor $F$. Finally, imagine we have estimated that the proportion of overall time that either processor spends using the device $H$ is $P$. The new execution time is derived as

$$EXECUTION\text{-TIME}(X) = EXECUTION\text{-TIME}(Y) \cdot ((1 - P) + (P/F)).$$

We can also work out the speed-up the improvement produced

$$\text{Speed-Up}(X \text{ over } Y) = \frac{EXECUTION\text{-TIME}(Y)}{EXECUTION\text{-TIME}(X)} = \frac{1}{(1-P)+(P/F)}.$$

To see that this is correct, consider the denominator of the fraction. The overall execution time of $Y$ will be the execution time of the unimproved portion of the design, which is $1 - P$, plus the execution time of the improved portion, which is $P/F$ since we improved it by the factor $F$. Finally, we divided the old execution time by the new execution time just like in our original formula.

Consider an example; imagine processor $Y$ takes ten seconds to execute a program. Then processor $X$ is designed to improve on $Y$ by using a new adder design, maybe it uses a carry look-ahead adder. The adder is used in about 20% of instructions and the new adder is four times as fast. Thus, we compute
\[
\text{SPEED-Up}(X \text{ over } Y) = \frac{1}{(1-P)+(P/F)} \\
= \frac{1}{(1-0.2)+(0.2/4)} \\
\sim 1.18.
\]

This might come as some surprise; our adder is four times as fast so why is the whole system not much faster?! This is the law of diminishing returns in effect: unless the adder is used more often, the improvement will not be capitalised on. As it is, processor X will take \( \sim 8.47 \) seconds using the new adder.

### 6.2 Further Reading

  *Performance Evaluation and Benchmarking.*  

- R.W. Hockney.  
  *The Science of Computer Benchmarking.*  

- R. Jain.  
  *The Art of Computer Systems Performance Analysis.*  

### 6.3 Example Questions

27. a. CPI and MIPS are both measures of processor performance. Explain how they are calculated and their advantages and disadvantages.

b. After some experimentation, someone shows that programs written for a given processor have the following instruction mix:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
</tr>
</tbody>
</table>

The addition of a new addressing mode and a better compiler means there are less accesses to memory required. The instruction mix after this change is
Calculate the overall CPI for programs that run on the processor before and after the change, and the speed-up that results from the change.

28. Outline the difficulties faced when writing benchmark software that is to collect information about the processor performance. Include at least one method that could be used to artificially increase the performance, i.e., “cheat” the benchmark.