2 Metrics of performance

'Time is a great teacher, but unfortunately it kills all its pupils.'

Hector Berlioz

2.1 What is a performance metric?

Before we can begin to understand any aspect of a computer system's performance, we must determine what things are interesting and useful to measure. The basic characteristics of a computer system that we typically need to measure are:

- a *count* of how many times an event occurs,
- the *duration* of some time interval, and
- the *size* of some parameter.

For instance, we may need to count how many times a processor initiates an input/output request. We may also be interested in how long each of these requests takes. Finally, it is probably also useful to determine the number of bits transmitted and stored.

From these types of measured values, we can derive the actual value that we wish to use to describe the performance of the system. This value is called a *performance metric*.

If we are interested specifically in the time, count, or size value measured, we can use that value directly as our performance metric. Often, however, we are interested in normalizing event counts to a common time basis to provide a speed metric such as operations executed per second. This type of metric is called a *rate metric* or *throughput* and is calculated by dividing the count of the number of events that occur in a given interval by the time interval over which the events occur. Since a rate metric is normalized to a common time basis, such as seconds, it is useful for comparing different measurements made over different time intervals.

Choosing an appropriate performance metric depends on the goals for the specific situation and the cost of gathering the necessary information. For

example, suppose that you need to choose between two different computer systems to use for a short period of time for one specific task, such as choosing between two systems to do some word processing for an afternoon. Since the penalty for being wrong in this case, that is, choosing the slower of the two machines, is very small, you may decide to use the processors' clock frequencies as the performance metric. Then you simply choose the system with the fastest clock. However, since the clock frequency is not a reliable performance metric (see Section 2.3.1), you would want to choose a better metric if you are trying to decide which system to buy when you expect to purchase hundreds of systems for your company. Since the consequences of being wrong are much larger in this case (you could lose your job, for instance!), you should take the time to perform a rigorous comparison using a better performance metric. This situation then begs the question of what constitutes a good performance metric.

2.2 Characteristics of a good performance metric

There are many different metrics that have been used to describe a computer system's performance. Some of these metrics are commonly used throughout the field, such as MIPS and MFLOPS (which are defined later in this chapter), whereas others are invented for new situations as they are needed. Experience has shown that not all of these metrics are 'good' in the sense that sometimes using a particular metric can lead to erroneous or misleading conclusions. Consequently, it is useful to understand the characteristics of a 'good' performance metric. This understanding will help when deciding which of the existing performance metrics to use for a particular situation, and when developing a new performance metric.

A performance metric that satisfies all of the following requirements is generally useful to a performance analyst in allowing accurate and detailed comparisons of different measurements. These criteria have been developed by observing the results of numerous performance analyses over many years. While they should not be considered absolute requirements of a performance metric, it has been observed that using a metric that does not satisfy these requirements can often lead the analyst to make erroneous conclusions.

1. **Linearity.** Since humans intuitively tend to think in linear terms, the value of the metric should be linearly proportional to the actual performance of the machine. That is, if the value of the metric changes by a certain ratio, the actual performance of the machine should change by the same ratio. This proportionality characteristic makes the metric intuitively appealing to most people. For example, suppose that you are upgrading your system to a system

whose speed metric (i.e. execution-rate metric) is twice as large as the same metric on your current system. You then would expect the new system to be able to run your application programs in half the time taken by your old system. Similarly, if the metric for the new system were three times larger than that of your current system, you would expect to see the execution times reduced to one-third of the original values.

Not all types of metrics satisfy this proportionally requirement. Logarithmic metrics, such as the dB scale used to describe the intensity of sound, for example, are nonlinear metrics in which an increase of one in the value of the metric corresponds to a factor of ten increase in the magnitude of the observed phenomenon. There is nothing inherently wrong with these types of nonlinear metrics, it is just that linear metrics tend to be more intuitively appealing when interpreting the performance of computer systems.

2. **Reliability.** A performance metric is considered to be *reliable* if system A always outperforms system B when the corresponding values of the metric for both systems indicate that system A should outperform system B. For example, suppose that we have developed a new performance metric called WIPS that we have designed to compare the performance of computer systems when running the class of word-processing application programs. We measure system A and find that it has a WIPS rating of 128, while system B has a WIPS rating of 97. We then can say that WIPS is a reliable performance metric for word-processing application programs if system A always outperforms system B when executing these types of applications.

   While this requirement would seem to be so obvious as to be unnecessary to state explicitly, several commonly used performance metrics do not in fact satisfy this requirement. The MIPS metric, for instance, which is described further in Section 2.3.2, is notoriously unreliable. Specifically, it is not unusual for one processor to have a higher MIPS rating than another processor while the second processor actually executes a specific program in less time than does the processor with the higher value of the metric. Such a metric is essentially useless for summarizing performance, and we say that it is unreliable.

3. **Repeatability.** A performance metric is *repeatable* if the same value of the metric is measured each time the same experiment is performed. Note that this also implies that a good metric is deterministic.

4. **Easiness of measurement.** If a metric is not easy to measure, it is unlikely that anyone will actually use it. Furthermore, the more difficult a metric is to measure directly, or to derive from other measured values, the more likely

it is that the metric will be determined incorrectly. The only thing worse than a bad metric is a metric whose value is measured incorrectly.

5. **Consistency.** A *consistent* performance metric is one for which the units of the metric and its precise definition are the same across different systems and different configurations of the same system. If the units of a metric are not consistent, it is impossible to use the metric to compare the performances of the different systems. While the necessity for this characteristic would also seem obvious, it is not satisfied by many popular metrics, such as MIPS (Section 2.3.2) and MFLOPS (Section 2.3.3).

6. **Independence.** Many purchasers of computer systems decide which system to buy by comparing the values of some commonly used performance metric. As a result, there is a great deal of pressure on manufacturers to design their machines to optimize the value obtained for that particular metric, and to influence the composition of the metric to their benefit. To prevent corruption of its meaning, a good metric should be *independent* of such outside influences.

2.3 Processor and system performance metrics

A wide variety of performance metrics has been proposed and used in the computer field. Unfortunately, many of these metrics are not good in the sense defined above, or they are often used and interpreted incorrectly. The following subsections describe many of these common metrics and evaluate them against the above characteristics of a good performance metric.

2.3.1 The clock rate

In many advertisements for computer systems, the most prominent indication of performance is often the frequency of the processor's central clock. The implication to the buyer is that a 250 MHz system must always be faster at solving the user's problem than a 200 MHz system, for instance. However, this performance metric completely ignores how much computation is actually accomplished in each clock cycle, it ignores the complex interactions of the processor with the memory subsystem and the input/output subsystem, and it ignores the not at all unlikely fact that the processor may not be the performance bottleneck.

Evaluating the clock rate against the characteristics for a good performance metric, we find that it is very repeatable (characteristic 3) since it is a constant for a given system, it is easy to measure (characteristic 4) since it is most likely stamped on the box, the value of MHz is precisely defined across all systems so that it is consistent (characteristic 5), and it is independent of any sort of

manufacturers' games (characteristic 6). However, the unavoidable shortcomings of using this value as a performance metric are that it is nonlinear (characteristic 1), and unreliable (characteristic 2). As many owners of personal computer systems can attest, buying a system with a faster clock in no way assures that their programs will run correspondingly faster. Thus, we conclude that the processor's clock rate is not a good metric of performance.

2.3.2 MIPS

A *throughput* or *execution-rate* performance metric is a measure of the amount of computation performed per unit time. Since rate metrics are normalized to a common basis, such as seconds, they are very useful for comparing relative speeds. For instance, a vehicle that travels at 50 m s^{-1} will obviously traverse more ground in a fixed time interval than will a vehicle traveling at 35 m s^{-1}.

The MIPS metric is an attempt to develop a rate metric for computer systems that allows a direct comparison of their speeds. While in the physical world speed is measured as the distance traveled per unit time, MIPS defines the computer system's unit of 'distance' as the execution of an instruction. Thus, MIPS, which is an acronym for *millions of instructions executed per second*, is defined to be

$$MIPS = \frac{n}{t_e \times 10^6} \tag{2.1}$$

where t_e is the time required to execute n total instructions.

Defining the unit of 'distance' in this way makes MIPS easy to measure (characteristic 4), repeatable (characteristic 3), and independent (characteristic 6). Unfortunately, it does not satisfy any of the other characteristics of a good performance metric. It is not linear since, like the clock rate, a doubling of the MIPS rate does not necessarily cause a doubling of the resulting performance. It also is neither reliable nor consistent since it really does not correlate well to performance at all.

The problem with MIPS as a performance metric is that different processors can do substantially different amounts of computation with a single instruction. For instance, one processor may have a branch instruction that branches after checking the state of a specified condition code bit. Another processor, on the other hand, may have a branch instruction that first decrements a specified count register, and then branches after comparing the resulting value in the register with zero. In the first case, a single instruction does one simple operation, whereas in the second case, one instruction actually performs several operations. The failing of the MIPS metric is that each instruction corresponds to one unit of 'distance,' even though in this example the second instruction actually performs more real computation. These differences in the amount of computation per-

formed by an instruction are at the heart of the differences between RISC and CISC processors and render MIPS essentially useless as a performance metric. Another derisive explanation of the MIPS acronym is *meaningless indicator of performance* since it is really no better a measure of overall performance than is the processor's clock frequency.

2.3.3 MFLOPS

The MFLOPS performance metric tries to correct the primary shortcoming of the MIPS metric by more precisely defining the unit of 'distance' traveled by a computer system when executing a program. MFLOPS, which is an acronym for *millions of floating-point operations executed per second*, defines an arithmetic operation on two floating-point (i.e. fractional) quantities to be the basic unit of 'distance.' MFLOPS is thus calculated as

$$MFLOPS = \frac{f}{t_e \times 10^6} \tag{2.2}$$

where f is the number of floating-point operations executed in t_e seconds. The MFLOPS metric is a definite improvement over the MIPS metric since the results of a floating-point computation are more clearly comparable across computer systems than is the execution of a single instruction. An important problem with this metric, however, is that the MFLOPS rating for a system executing a program that performs no floating-point calculations is exactly zero. This program may actually be performing very useful operations, though, such as searching a database or sorting a large set of records.

A more subtle problem with MFLOPS is agreeing on exactly how to count the number of floating-point operations in a program. For instance, many of the Cray vector computer systems performed a floating-point division operation using successive approximations involving the reciprocal of the denominator and several multiplications. Similarly, some processors can calculate transcendental functions, such as sin, cos, and log, in a single instruction, while others require several multiplications, additions, and table look-ups. Should these operations be counted as a single floating-point operation or multiple floating-point operations? The first method would intuitively seem to make the most sense. The second method, however, would increase the value of f in the above calculation of the MFLOPS rating, thereby artificially inflating its value. This flexibility in counting the total number of floating-point operations causes MFLOPS to violate characteristic 6 of a good performance metric. It is also unreliable (characteristic 2) and inconsistent (characteristic 5).

2.3.4 SPEC

To standardize the definition of the actual result produced by a computer system in 'typical' usage, several computer manufacturers banded together to form the System Performance Evaluation Cooperative (SPEC). This group identified a set of integer and floating-point benchmark programs that was intended to reflect the way most workstation-class computer systems were actually used. Additionally, and, perhaps, most importantly, they also standardized the methodology for measuring and reporting the performance obtained when executing these programs.

The methodology defined consists of the following key steps.

1. Measure the time required to execute each program in the set on the system being tested.
2. Divide the time measured for each program in the first step by the time required to execute each program on a standard basis machine to normalize the execution times.
3. Average together all of these normalized values using the geometric mean (see Section 3.3.4) to produce a single-number performance metric.

While the SPEC methodology is certainly more rigorous than is using MIPS or MFLOPS as a measure of performance, it still produces a problematic performance metric. One shortcoming is that averaging together the individual normalized results with the geometric mean produces a metric that is not linearly related to a program's actual execution time. Thus, the SPEC metric is not intuitive (characteristic 1). Furthermore, and more importantly, it has been shown to be an unreliable metric (characteristic 2) in that a given program may execute faster on a system that has a lower SPEC rating than it does on a competing system with a higher rating.

Finally, although the defined methodology appears to make the metric independent of outside influences (characteristic 6), it is actually subject to a wide range of tinkering. For example, many compiler developers have used these benchmarks as practice programs, thereby tuning their optimizations to the characteristics of this collection of applications. As a result, the execution times of the collection of programs in the SPEC suite can be quite sensitive to the particular selection of optimization flags chosen when the program is compiled. Also, the selection of specific programs that comprise the SPEC suite is determined by a committee of representatives from the manufacturers within the cooperative. This committee is subject to numerous outside pressures since each manufacturer has a strong interest in advocating application programs that will perform well on their machines. Thus, while SPEC is a significant step in the right direction towards defining a good performance metric, it still falls short of the goal.

2.3.5 QUIPS

The QUIPS metric, which was developed in conjunction with the HINT benchmark program, is a fundamentally different type of performance metric. (The details of the HINT benchmark and the precise definition of QUIPS are given in Section 7.2.3). Instead of defining the *effort* expended to reach a certain result as the measure of what is accomplished, the QUIPS metric defines the *quality of the solution* as a more meaningful indication of a user's final goal. The quality is rigorously defined on the basis of mathematical characteristics of the problem being solved. Dividing this measure of solution quality by the time required to achieve that level of quality produces QUIPS, or *quality improvements per second*.

This new performance metric has several of the characteristics of a good performance metric. The mathematically precise definition of 'quality' for the defined problem makes this metric insensitive to outside influences (characteristic 6) and makes it entirely self-consistent when it is ported to different machines (characteristic 5). It is also easily repeatable (characteristic 3) and it is linear (characteristic 1) since, for the particular problem chosen for the HINT benchmark, the resulting measure of quality is linearly related to the time required to obtain the solution.

Given the positive aspects of this metric, it still does present a few potential difficulties when used as a general-purpose performance metric. The primary potential difficulty is that it need not always be a reliable metric (characteristic 2) due to its narrow focus on floating-point and memory system performance. It is generally a very good metric for predicting how a computer system will perform when executing numerical programs. However, it does not exercise some aspects of a system that are important when executing other types of application programs, such as the input/output subsystem, the instruction cache, and the operating system's ability to multiprogram, for instance. Furthermore, while the developers have done an excellent job of making the HINT benchmark easy to measure (characteristic 4) and portable to other machines, it is difficult to change the quality definition. A new problem must be developed to focus on other aspects of a system's performance since the definition of quality is tightly coupled to the problem being solved. Developing a new problem to more broadly exercise the system could be a difficult task since it must maintain all of the characteristics described above.

Despite these difficulties, QUIPS is an important new type of metric that rigorously defines interesting aspects of performance while providing enough flexibility to allow new and unusual system architectures to demonstrate their capabilities. While it is not a completely general-purpose metric, it should prove to be very useful in measuring a system's numerical processing capabilities.

It also should be a strong stimulus for greater rigor in defining future performance metrics.

2.3.6 Execution time

Since we are ultimately interested in how quickly a given program is executed, the fundamental performance metric of any computer system is the time required to execute a given application program. Quite simply, the system that produces the smallest total execution time for a given application program has the highest performance. We can compare times directly, or use them to derive appropriate rates. However, without a precise and accurate measure of time, it is impossible to analyze or compare most any system performance characteristics. Consequently, it is important to know how to measure the execution time of a program, or a portion of a program, and to understand the limitations of the measuring tool.

The basic technique for measuring time in a computer system is analogous to using a stopwatch to measure the time required to perform some event. Unlike a stopwatch that begins measuring time from 0, however, a computer system typically has an internal counter that simply counts the number of clock ticks that have occurred since the system was first turned on. (See also Section 6.2.) A time interval then is measured by reading the value of the counter at the start of the event to be timed and again at the end of the event. The elapsed time is the difference between the two count values multiplied by the period of the clock ticks.

As an example, consider the program example shown in Figure 2.1. In this example, the `init_timer()` function initializes the data structures used to access the system's timer. This timer is a simple counter that is incremented continuously by a clock with a period defined in the variable `clock_cycle`. Reading the address pointed to by the variable `read_count` returns the current count value of the timer.

To begin timing a portion of a program, the current value in the timer is read and stored in `start_count`. At the end of the portion of the program being timed, the timer value is again read and stored in `end_count`. The difference between these two values is the total number of clock ticks that occurred during the execution of the event being measured. The total time required to execute this event is this number of clock ticks multiplied by the period of each tick, which is stored in the constant `clock_cycle`.

This technique for measuring the elapsed execution time of any selected portion of a program is often referred to as the *wall clock* time since it measures the total time that a user would have to wait to obtain the results produced by the program. That is, the measurement includes the time spent waiting for input/

```
main()
{
    int i;
    float a;

    init_timer();

    /* Read the starting time. */
    start_count = read_count;

    /* Stuff to be measured */
    for (i=0;i< 1000;i++){
        a = i * a / 10;
    }

    /* Read the ending time. */
    end_count = read_count;

    elapsed_time = (end_count - start_count) * clock_cycle;
}
```

Figure 2.1. An example program showing how to measure the execution time of a portion of a program.

output operations to complete, memory paging, and other system operations performed on behalf of this application, all of which are integral components of the program's execution. However, when the system being measured is time-shared so that it is not dedicated to the execution of this one application program, this elapsed execution time also includes the time the application spends waiting while other users' applications execute.

Many researchers have argued that including this time-sharing overhead in the program's total execution time is unfair. Instead, they advocate measuring performance using the total time the processor actually spends executing the program, called the total *CPU time*. This time does not include the time the program is context switched-out while another application runs. Unfortunately, using only this CPU time as the performance metric ignores the waiting time that is inherent to the application as well as the time spent waiting on other programs. A better solution is to report both the CPU time and the total execution time so the reader can determine the significance of the time-sharing interference. The point is to be explicit about what information you are actually reporting to allow the reader to decide for themselves how believable your results are.

In addition to system-overhead effects, the measured execution time of an application program can vary significantly from one run to another since the program must contend with random events, such as the execution of background operating system tasks, different virtual-to-physical page mappings and cache mappings from explicitly random replacement policies, variable system load in a time-shared system, and so forth. As a result, a program's execution time is nondeterministic. It is important, then, to measure a program's total elapsed execution time several times and report at least the mean and variance of the times. Errors in measurements, along with appropriate statistical techniques to quantify them, are discussed in more detail in Chapter 4.

When it is measured as described above, the elapsed (wall clock) time measurement produces a performance metric that is intuitive, reliable, repeatable, easy to measure, consistent across systems, and independent of outside influences. Thus, since it satisfies all of the characteristics of a good performance metric, program execution time is one of the best metrics to use when analyzing computer system performance.

2.4 Other types of performance metrics

In addition to the processor-centric metrics described above, there are many other metrics that are commonly used in performance analysis. For instance, the system *response time* is the amount of time that elapses from when a user submits a request until the result is returned from the system. This metric is often used in analyzing the performance of online transaction-processing systems, for example. System *throughput* is a measure of the number of jobs or operations that are completed per unit time. The performance of a real-time video-processing system, for instance, may be measured in terms of the number of video frames that can be processed per second. The *bandwidth* of a communication network is a throughput measure that quantifies the number of bits that can be transmitted across the network per second. Many other *ad hoc* performance metrics are defined by performance analysts to suit the specific needs of the problem or system being studied.

2.5 Speedup and relative change

Speedup and *relative change* are useful metrics for comparing systems since they normalize performance to a common basis. Although these metrics are defined in terms of throughput or speed metrics, they are often calculated directly from execution times, as described below.

Speedup. The *speedup* of system 2 with respect to system 1 is defined to be a value $S_{2,1}$ such that $R_2 = S_{2,1}R_1$, where R_1 and R_2 are the 'speed' metrics being compared. Thus, we can say that system 2 is $S_{2,1}$ times faster than system 1. Since a speed metric is really a rate metric (i.e. throughput), $R_1 = D_1/T_1$, where D_1 is analogous to the 'distance traveled' in time T_1 by the application program when executing on system 1. Similarly, $R_2 = D_2/T_2$. Assuming that the 'distance traveled' by each system is the same, $D_1 = D_2 = D$, giving the following definition for speedup:

$$\textit{Speedup of system 2 w.r.t. system 1} = S_{2,1} = \frac{R_2}{R_1} = \frac{D/T_2}{D/T_1} = \frac{T_1}{T_2}. \tag{2.3}$$

If system 2 is faster than system 1, then $T_2 < T_1$ and the speedup ratio will be larger than 1. If system 2 is slower than system 1, however, the speedup ratio will be less than 1. This situation is often referred to as a *slowdown* instead of a speedup.

Relative change. Another technique for normalizing performance is to express the performance of a system as a percent change *relative* to the performance of another system. We again use the throughput metrics R_1 and R_2 as measures of the speeds of systems 1 and 2, respectively. The relative change of system 2 with respect to system 1, denoted $\Delta_{2,1}$, (that is, using system 1 as the basis) is then defined to be

$$\textit{Relative change of system 2 w.r.t. system 1} = \Delta_{2,1} = \frac{R_2 - R_1}{R_1}. \tag{2.4}$$

Again assuming that the execution time of each system is measured when executing the same program, the 'distance traveled' by each system is the same so that $R_1 = D/T_1$ and $R_2 = D/T_2$. Thus,

$$\Delta_{2,1} = \frac{R_2 - R_1}{R_1} = \frac{D/T_2 - D/T_1}{D/T_1} = \frac{T_1 - T_2}{T_2} = S_{2,1} - 1. \tag{2.5}$$

Typically, the value of $\Delta_{2,1}$ is multiplied by 100 to express the relative change as a percentage with respect to a given basis system. This definition of relative change will produce a positive value if system 2 is faster than system 1, whereas a negative value indicates that the basis system is faster.

Example. As an example of how to apply these two different normalization techniques, the speedup and relative change of the systems shown in Table 2.1 are found using system 1 as the basis. From the raw execution times, we can easily see that system 4 is the fastest, followed by systems 2, 1, and 3, in that order. However, the speedup values give us a more precise indication of exactly how much faster one system is than the others. For instance, system 2 has a

Table 2.1. An example of calculating speedup and relative change using system 1 as the basis

System x	Execution time T_x (s)	Speedup $S_{x,1}$	Relative change $\Delta_{x,1}$ (%)
1	480	1	0
2	360	1.33	+33
3	540	0.89	−11
4	210	2.29	+129

speedup of 1.33 compared with system 1 or, equivalently, it is 33% faster. System 4 has a speedup ratio of 2.29 compared with system 1 (or it is 129% faster). We also see that system 3 is actually 11% slower than system 1, giving it a slowdown factor of 0.89. ◇

2.6 Means versus ends metrics

One of the most important characteristics of a performance metric is that it be reliable (characteristic 2). One of the problems with many of the metrics discussed above that makes them unreliable is that they measure what was done *whether or not it was useful*. What makes a performance metric reliable, however, is that it accurately and consistently measures *progress towards a goal*. Metrics that measure what was done, useful or not, have been called *means-based* metrics whereas *ends-based* metrics measure what is actually accomplished.

To obtain a feel for the difference between these two types of metrics, consider the vector dot-product routine shown in Figure 2.2. This program executes N floating-point addition and multiplication operations for a total of $2N$ floating-point operations. If the time required to execute one addition is t_+ cycles and one multiplication requires t_* cycles, the total time required to execute this program is $t_1 = N(t_+ + t_*)$ cycles. The resulting execution rate then is

```
s = 0;
for (i = 1; i < N; i++)
   s = s + x[i] * y[i];
```

Figure 2.2. A vector dot-product example program.

$$R_1 = \frac{2N}{N(t_+ + t_*)} = \frac{2}{t_+ + t_*} \text{FLOPS/cycle.} \tag{2.6}$$

Since there is no need to perform the addition or multiplication operations for elements whose value is zero, it may be possible to reduce the total execution time if many elements of the two vectors are zero. Figure 2.3 shows the example from Figure 2.2 modified to perform the floating-point operations only for those nonzero elements. If the conditional `if` statement requires t_{if} cycles to execute, the total time required to execute this program is $t_2 = N[t_{\text{if}} + f(t_+ + t_*)]$ cycles, where f is the fraction of N for which both `x[i]` and `y[i]` are nonzero. Since the total number of additions and multiplications executed in this case is $2Nf$, the execution rate for this program is

$$R_2 = \frac{2Nf}{N[t_{\text{if}} + f(t_+ + t_*)]} = \frac{2f}{t_{\text{if}} + f(t_+ + t_*)} \text{FLOPS/cycle.} \tag{2.7}$$

If t_{if} is four cycles, t_+ is five cycles, t_* is ten cycles, f is 10%, and the processor's clock rate is 250 MHz (i.e. one cycle is 4 ns), then $t_1 = 60N$ ns and $t_2 = N[4 + 0.1(5 + 10)] \times 4$ ns $= 22N$ ns. The speedup of program 2 relative to program 1 then is found to be $S_{2,1} = 60N/22N = 2.73$.

Calculating the execution rates realized by each program with these assumptions produces $R_1 = 2/(60 \text{ ns}) = 33$ MFLOPS and $R_2 = 2(0.1)/(22 \text{ ns}) = 9.09$ MFLOPS. Thus, even though we have reduced the total execution time from $t_1 = 60N$ ns to $t_2 = 22N$ ns, the means-based metric (MFLOPS) shows that program 2 is 72% slower than program 1. The ends-based metric (execution time), however, shows that program 2 is actually 173% faster than program 1. We reach completely different conclusions when using these two different types of metrics because the means-based metric unfairly gives program 1 credit for all of the useless operations of multiplying and adding zero. This example highlights the danger of using the wrong metric to reach a conclusion about computer-system performance.

```
s = 0;
for (i = 1; i < N; i++)
    if (x[i] != 0 && y[i] != 0)
        s = s + x[i] * y[i];
```

Figure 2.3. The vector dot-product example program of Figure 2.2 modified to calculate only nonzero elements.

2.7 Summary

Fundamental to measuring computer-systems performance is defining an appropriate metric. This chapter identified several characteristics or criteria that are important for a 'good' metric of performance. Several common performance metrics were then introduced and analyzed in the context of these criteria. The definitions of speedup and relative change were also introduced. Finally, the concepts of ends-based versus means-based metrics were presented to clarify what actually causes a metric to be useful in capturing the actual performance of a computer system.

2.8 For further reading

- The following paper argues strongly for total execution time as the best measure of performance:

  James E. Smith, 'Characterizing Computer Performance with a Single Number,' *Communications of the ACM*, October 1988, pp. 1202–1206.

- The QUIPS metric is described in detail in the following paper, which also introduced the idea of means-based versus ends-based metrics:

  J. L. Gustafson and Q. O. Snell, 'HINT: A New Way to Measure Computer Performance,' *Hawaii International Conference on System Sciences,* 1995, pp. II:392–401.

- Some of the characteristics of the SPEC metric are discussed in the following papers:

  Ran Giladi and Niv Ahituv, 'SPEC as a Performance Evaluation Measure,' *IEEE Computer,* Vol. 28, No. 8, August 1995, pp. 33–42.

  Nikki Mirghafori, Margret Jacoby, and David Patterson, 'Truth in SPEC Benchmarks,' *ACM Computer Architecture News,* Vol. 23, No. 5, December 1995, pp. 34–42.

- Parallel computing systems are becoming more common. They present some interesting performance measurement problems, though, as discussed in

  Lawrence A. Crowl, 'How to Measure, Present, and Compare Parallel Performance,' *IEEE Parallel and Distributed Technology,* Spring 1994, pp. 9–25.

2.9 Exercises

1. (a) Write a simple benchmark program to estimate the maximum effective MIPS rating of a computer system. Use your program to rank the performance of three different, but roughly comparable, computer systems.
   (b) Repeat part (a) using the maximum effective MFLOPS rating as the metric of performance.
   (c) Compare the rankings obtained in parts (a) and (b) with the ranking obtained by comparing the clock frequencies of the different systems.
   (d) Finally, compare your rankings with those published by authors using some standard benchmark programs, such as those available on the SPEC website.
2. What makes a performance metric 'reliable?'
3. Classify each of the following metrics as being either means-based or ends-based; MIPS, MFLOPS, execution time, bytes of available memory, quality of a final answer, arithmetic precision, system cost, speedup, and reliability of an answer.
4. Devise an experiment to determine the following performance metrics for a computer system.
   (a) The effective memory bandwidth between the processor and the data cache if all memory references are cache hits.
   (b) The effective memory bandwidth if all memory references are cache misses.
5. What are the key differences between 'wall clock time' and 'CPU time?' Under what conditions should each one be used? Is it possible for these two different times to be the same?
6. The execution time required to read the current time from an interval counter is a minimum of at least one memory-read operation to obtain the current time value and one memory-write operation to store the value for later use. In some cases, it may additionally include a subroutine call and return operation. How does this timer 'overhead' affect the time measured when using such an interval timer to determine the duration of some event, such as the total execution time of a program?
7. Calculate the speedup and relative change of the four systems shown in Table 2.1 when using System 4 as the basis. How do your newly calculated values affect the relative rankings of the four systems?